JUNIPER NETWORKS

Junos® OS

Configuration and Operations Automation Guide

Release

10.4

Published: 2010-10-08

Juniper Networks, Inc.
1194 North Mathilda Avenue
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

Junos® OS Configuration and Operations Automation Guide
Release 10.4

Writing: Michael Scruggs, Brenda Wilden
Editing: Sonia Saruba, Joanne McClintock, Nancy Kurahashi
Illustration: Faith Bradford
Cover Design: Edmonds Design

Revision History
October 2010—R1 Junos 10.4

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

YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. The Junos OS has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

END USER LICENSE AGREEMENT

READ THIS END USER LICENSE AGREEMENT ("AGREEMENT") BEFORE DOWNLOADING, INSTALLING, OR USING THE SOFTWARE. BY DOWNLOADING, INSTALLING, OR USING THE SOFTWARE OR OTHERWISE EXPRESSING YOUR AGREEMENT TO THE TERMS CONTAINED HEREIN, YOU (AS CUSTOMER OR IF YOU ARE NOT THE CUSTOMER, AS A REPRESENTATIVE/AGENT AUTHORIZED TO BIND THE CUSTOMER) CONSENT TO BE BOUND BY THIS AGREEMENT. IF YOU DO NOT OR CANNOT AGREE TO THE TERMS CONTAINED HEREIN, THEN (A) DO NOT DOWNLOAD, INSTALL, OR USE THE SOFTWARE, AND (B) YOU MAY CONTACT JUNIPER NETWORKS REGARDING LICENSE TERMS.

1. **The Parties.** The parties to this Agreement are (i) Juniper Networks, Inc. (if the Customer's principal office is located in the Americas) or Juniper Networks (Cayman) Limited (if the Customer's principal office is located outside the Americas) (such applicable entity being referred to herein as "Juniper"), and (ii) the person or organization that originally purchased from Juniper or an authorized Juniper reseller the applicable license(s) for use of the Software ("Customer") (collectively, the "Parties").

2. **The Software.** In this Agreement, "Software" means the program modules and features of the Juniper or Juniper-supplied software, for which Customer has paid the applicable license or support fees to Juniper or an authorized Juniper reseller, or which was embedded by Juniper in equipment which Customer purchased from Juniper or an authorized Juniper reseller. "Software" also includes updates, upgrades and new releases of such software. "Embedded Software" means Software which Juniper has embedded in or loaded onto the Juniper equipment and any updates, upgrades, additions or replacements which are subsequently embedded in or loaded onto the equipment.

3. **License Grant.** Subject to payment of the applicable fees and the limitations and restrictions set forth herein, Juniper grants to Customer a non-exclusive and non-transferable license, without right to sublicense, to use the Software, in executable form only, subject to the following use restrictions:

a. Customer shall use Embedded Software solely as embedded in, and for execution on, Juniper equipment originally purchased by Customer from Juniper or an authorized Juniper reseller.

b. Customer shall use the Software on a single hardware chassis having a single processing unit, or as many chassis or processing units for which Customer has paid the applicable license fees; provided, however, with respect to the Steel-Belted Radius or Odyssey Access Client software only, Customer shall use such Software on a single computer containing a single physical random access memory space and containing any number of processors. Use of the Steel-Belted Radius or IMS AAA software on multiple computers or virtual machines (e.g., Solaris zones) requires multiple licenses, regardless of whether such computers or virtualizations are physically contained on a single chassis.

c. Product purchase documents, paper or electronic user documentation, and/or the particular licenses purchased by Customer may specify limits to Customer's use of the Software. Such limits may restrict use to a maximum number of seats, registered endpoints, concurrent users, sessions, calls, connections, subscribers, clusters, nodes, realms, devices, links, ports or transactions, or require the purchase of separate licenses to use particular features, functionalities, services, applications, operations, or capabilities, or provide throughput, performance, configuration, bandwidth, interface, processing, temporal, or geographical limits. In addition, such limits may restrict the use of the Software to managing certain kinds of networks or require the Software to be used only in conjunction with other specific Software. Customer's use of the Software shall be subject to all such limitations and purchase of all applicable licenses.

d. For any trial copy of the Software, Customer's right to use the Software expires 30 days after download, installation or use of the Software. Customer may operate the Software after the 30-day trial period only if Customer pays for a license to do so. Customer may not extend or create an additional trial period by re-installing the Software after the 30-day trial period.

e. The Global Enterprise Edition of the Steel-Belted Radius software may be used by Customer only to manage access to Customer's enterprise network. Specifically, service provider customers are expressly prohibited from using the Global Enterprise Edition of the Steel-Belted Radius software to support any commercial network access services.

The foregoing license is not transferable or assignable by Customer. No license is granted herein to any user who did not originally purchase the applicable license(s) for the Software from Juniper or an authorized Juniper reseller.

4. **Use Prohibitions.** Notwithstanding the foregoing, the license provided herein does not permit the Customer to, and Customer agrees not to and shall not: (a) modify, unbundle, reverse engineer, or create derivative works based on the Software; (b) make unauthorized copies of the Software (except as necessary for backup purposes); (c) rent, sell, transfer, or grant any rights in and to any copy of the Software, in any form, to any third party; (d) remove any proprietary notices, labels, or marks on or in any copy of the Software or any product in which the Software is embedded; (e) distribute any copy of the Software to any third party, including as may be embedded in Juniper equipment sold in the secondhand market; (f) use any 'locked' or key-restricted feature, function, service, application, operation, or capability without first purchasing the applicable license(s) and obtaining a valid key from Juniper, even if such feature, function, service, application, operation, or capability is enabled without a key; (g) distribute any key for the Software provided by Juniper to any third party; (h) use the

Software in any manner that extends or is broader than the uses purchased by Customer from Juniper or an authorized Juniper reseller; (i) use Embedded Software on non-Juniper equipment; (j) use Embedded Software (or make it available for use) on Juniper equipment that the Customer did not originally purchase from Juniper or an authorized Juniper reseller; (k) disclose the results of testing or benchmarking of the Software to any third party without the prior written consent of Juniper; or (l) use the Software in any manner other than as expressly provided herein.

5. **Audit.** Customer shall maintain accurate records as necessary to verify compliance with this Agreement. Upon request by Juniper, Customer shall furnish such records to Juniper and certify its compliance with this Agreement.

6. **Confidentiality.** The Parties agree that aspects of the Software and associated documentation are the confidential property of Juniper. As such, Customer shall exercise all reasonable commercial efforts to maintain the Software and associated documentation in confidence, which at a minimum includes restricting access to the Software to Customer employees and contractors having a need to use the Software for Customer's internal business purposes.

7. **Ownership.** Juniper and Juniper's licensors, respectively, retain ownership of all right, title, and interest (including copyright) in and to the Software, associated documentation, and all copies of the Software. Nothing in this Agreement constitutes a transfer or conveyance of any right, title, or interest in the Software or associated documentation, or a sale of the Software, associated documentation, or copies of the Software.

8. **Warranty, Limitation of Liability, Disclaimer of Warranty.** The warranty applicable to the Software shall be as set forth in the warranty statement that accompanies the Software (the "Warranty Statement"). Nothing in this Agreement shall give rise to any obligation to support the Software. Support services may be purchased separately. Any such support shall be governed by a separate, written support services agreement. TO THE MAXIMUM EXTENT PERMITTED BY LAW, JUNIPER SHALL NOT BE LIABLE FOR ANY LOST PROFITS, LOSS OF DATA, OR COSTS OR PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY SPECIAL, INDIRECT, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THIS AGREEMENT, THE SOFTWARE, OR ANY JUNIPER OR JUNIPER-SUPPLIED SOFTWARE. IN NO EVENT SHALL JUNIPER BE LIABLE FOR DAMAGES ARISING FROM UNAUTHORIZED OR IMPROPER USE OF ANY JUNIPER OR JUNIPER-SUPPLIED SOFTWARE. EXCEPT AS EXPRESSLY PROVIDED IN THE WARRANTY STATEMENT TO THE EXTENT PERMITTED BY LAW, JUNIPER DISCLAIMS ANY AND ALL WARRANTIES IN AND TO THE SOFTWARE (WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE), INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT DOES JUNIPER WARRANT THAT THE SOFTWARE, OR ANY EQUIPMENT OR NETWORK RUNNING THE SOFTWARE, WILL OPERATE WITHOUT ERROR OR INTERRUPTION, OR WILL BE FREE OF VULNERABILITY TO INTRUSION OR ATTACK. In no event shall Juniper's or its suppliers' or licensors' liability to Customer, whether in contract, tort (including negligence), breach of warranty, or otherwise, exceed the price paid by Customer for the Software that gave rise to the claim, or if the Software is embedded in another Juniper product, the price paid by Customer for such other product. Customer acknowledges and agrees that Juniper has set its prices and entered into this Agreement in reliance upon the disclaimers of warranty and the limitations of liability set forth herein, that the same reflect an allocation of risk between the Parties (including the risk that a contract remedy may fail of its essential purpose and cause consequential loss), and that the same form an essential basis of the bargain between the Parties.

9. **Termination.** Any breach of this Agreement or failure by Customer to pay any applicable fees due shall result in automatic termination of the license granted herein. Upon such termination, Customer shall destroy or return to Juniper all copies of the Software and related documentation in Customer's possession or control.

10. **Taxes.** All license fees payable under this agreement are exclusive of tax. Customer shall be responsible for paying Taxes arising from the purchase of the license, or importation or use of the Software. If applicable, valid exemption documentation for each taxing jurisdiction shall be provided to Juniper prior to invoicing, and Customer shall promptly notify Juniper if their exemption is revoked or modified. All payments made by Customer shall be net of any applicable withholding tax. Customer will provide reasonable assistance to Juniper in connection with such withholding taxes by promptly: providing Juniper with valid tax receipts and other required documentation showing Customer's payment of any withholding taxes; completing appropriate applications that would reduce the amount of withholding tax to be paid; and notifying and assisting Juniper in any audit or tax proceeding related to transactions hereunder. Customer shall comply with all applicable tax laws and regulations, and Customer will promptly pay or reimburse Juniper for all costs and damages related to any liability incurred by Juniper as a result of Customer's non-compliance or delay with its responsibilities herein. Customer's obligations under this Section shall survive termination or expiration of this Agreement.

11. **Export.** Customer agrees to comply with all applicable export laws and restrictions and regulations of any United States and any applicable foreign agency or authority, and not to export or re-export the Software or any direct product thereof in violation of any such restrictions, laws or regulations, or without all necessary approvals. Customer shall be liable for any such violations. The version of the Software supplied to Customer may contain encryption or other capabilities restricting Customer's ability to export the Software without an export license.

12. **Commercial Computer Software.** The Software is "commercial computer software" and is provided with restricted rights. Use, duplication, or disclosure by the United States government is subject to restrictions set forth in this Agreement and as provided in DFARS 227.7201 through 227.7202-4, FAR 12.212, FAR 27.405(b)(2), FAR 52.227-19, or FAR 52.227-14(ALT III) as applicable.

13. **Interface Information.** To the extent required by applicable law, and at Customer's written request, Juniper shall provide Customer with the interface information needed to achieve interoperability between the Software and another independently created program, on payment of applicable fee, if any. Customer shall observe strict obligations of confidentiality with respect to such information and shall use such information in compliance with any applicable terms and conditions upon which Juniper makes such information available.

14. **Third Party Software.** Any licensor of Juniper whose software is embedded in the Software and any supplier of Juniper whose products or technology are embedded in (or services are accessed by) the Software shall be a third party beneficiary with respect to this Agreement, and such licensor or vendor shall have the right to enforce this Agreement in its own name as if it were Juniper. In addition, certain third party software may be provided with the Software and is subject to the accompanying license(s), if any, of its respective owner(s). To the extent portions of the Software are distributed under and subject to open source licenses obligating Juniper to make the source code for such portions publicly available (such as the GNU General Public License ("GPL") or the GNU Library General Public License ("LGPL")), Juniper will make such source code portions (including Juniper modifications, as appropriate) available upon request for a period of up to three years from the date of distribution. Such request can be made in writing to Juniper Networks, Inc., 1194 N. Mathilda Ave., Sunnyvale, CA 94089, ATTN: General Counsel. You may obtain a copy of the GPL at http://www.gnu.org/licenses/gpl.html, and a copy of the LGPL at http://www.gnu.org/licenses/lgpl.html .

15. **Miscellaneous.** This Agreement shall be governed by the laws of the State of California without reference to its conflicts of laws principles. The provisions of the U.N. Convention for the International Sale of Goods shall not apply to this Agreement. For any disputes arising under this Agreement, the Parties hereby consent to the personal and exclusive jurisdiction of, and venue in, the state and federal courts within Santa Clara County, California. This Agreement constitutes the entire and sole agreement between Juniper and the Customer with respect to the Software, and supersedes all prior and contemporaneous agreements relating to the Software, whether oral or written (including any inconsistent terms contained in a purchase order), except that the terms of a separate written agreement executed by an authorized Juniper representative and Customer shall govern to the extent such terms are inconsistent or conflict with terms contained herein. No modification to this Agreement nor any waiver of any rights hereunder shall be effective unless expressly assented to in writing by the party to be charged. If any portion of this Agreement is held invalid, the Parties agree that such invalidity shall not affect the validity of the remainder of this Agreement. This Agreement and associated documentation has been written in the English language, and the Parties agree that the English version will govern. (For Canada: Les parties aux présentés confirment leur volonté que cette convention de même que tous les documents y compris tout avis qui s'y rattaché, soient redigés en langue anglaise. (Translation: The parties confirm that this Agreement and all related documentation is and will be in the English language)).

Abbreviated Table of Contents

Table of Contents

List of Figures

List of Tables

About This Guide

This preface provides the following guidelines for using the *Junos® OS Configuration and Operations Automation Guide*:

Junos Documentation and Release Notes

For a list of related Junos documentation, see http://www.juniper.net/techpubs/software/junos/ .

If the information in the latest release notes differs from the information in the documentation, follow the *Junos Release Notes.*

To obtain the most current version of all Juniper Networks® technical documentation, see the product documentation page on the Juniper Networks website at http://www.juniper.net/techpubs/.

Juniper Networks supports a technical book program to publish books by Juniper Networks engineers and subject matter experts with book publishers around the world. These books go beyond the technical documentation to explore the nuances of network architecture, deployment, and administration using the Junos operating system (Junos OS) and Juniper Networks devices. In addition, the Juniper Networks Technical Library, published in conjunction with O'Reilly Media, explores improving network security, reliability, and availability using Junos OS configuration techniques. All the books are for sale at technical bookstores and book outlets around the world. The current list can be viewed at http://www.juniper.net/books .

Objectives

This guide provides an overview, instructions for using, and examples of Junos automation and the self-diagnosis features of the Junos OS. Junos automation scripts, which include commit scripts, operation scripts, and event scripts, are based on Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX), the Junos Extensible Markup Language (XML) application programming interface (API), and the Junos XML Management Protocol. This guide also explains how to use commit script macros to provide simplified aliases for frequently used configuration statements and how to configure diagnostic event policies and actions associated with each policy.

NOTE: For additional information about Junos OS—either corrections to or information that might have been omitted from this guide—see the software release notes at http://www.juniper.net/.

Audience

This guide is designed for network administrators who are configuring and monitoring a Juniper Networks M Series, MX Series, T Series, EX Series, or J Series router or switch.

To use this guide, you need a broad understanding of networks in general, the Internet in particular, networking principles, and network configuration. You must also be familiar with one or more of the following Internet routing protocols:

- Border Gateway Protocol (BGP)
- Distance Vector Multicast Routing Protocol (DVMRP)
- Intermediate System-to-Intermediate System (IS-IS)
- Internet Control Message Protocol (ICMP) router discovery
- Internet Group Management Protocol (IGMP)
- Multiprotocol Label Switching (MPLS)
- Open Shortest Path First (OSPF)
- Protocol-Independent Multicast (PIM)
- Resource Reservation Protocol (RSVP)
- Routing Information Protocol (RIP)
- Simple Network Management Protocol (SNMP)

Personnel operating the equipment must be trained and competent; must not conduct themselves in a careless, willfully negligent, or hostile manner; and must abide by the instructions provided by the documentation.

Supported Platforms

For the features described in this manual, Junos OS currently supports the following platforms:

- EX Series
- J Series
- M Series
- MX Series
- SRX Series
- T Series

Using the Indexes

This reference contains two indexes: a standard index with topic entries, and an index of commands.

Using the Examples in This Manual

If you want to use the examples in this manual, you can use the **load merge** or the **load merge relative** command. These commands cause the software to merge the incoming configuration into the current candidate configuration. If the example configuration contains the top level of the hierarchy (or multiple hierarchies), the example is a *full example*. In this case, use the **load merge** command.

If the example configuration does not start at the top level of the hierarchy, the example is a *snippet*. In this case, use the **load merge relative** command. These procedures are described in the following sections.

Merging a Full Example

To merge a full example, follow these steps:

1. From the HTML or PDF version of the manual, copy a configuration example into a text file, save the file with a name, and copy the file to a directory on your routing platform.

   For example, copy the following configuration to a file and name the file **ex-script.conf**. Copy the **ex-script.conf** file to the **/var/tmp** directory on your routing platform.

   ```
   system {
     scripts {
       commit {
         file ex-script.xsl;
       }
     }
   }
   interfaces {
     fxp0 {
   ```

```
        disable;
        unit 0 {
          family inet {
            address 10.0.0.1/24;
          }
        }
      }
    }
```

2. Merge the contents of the file into your routing platform configuration by issuing the **load merge** configuration mode command:

```
[edit]
user@host# load merge /var/tmp/ex-script.conf
load complete
```

Merging a Snippet

To merge a snippet, follow these steps:

1. From the HTML or PDF version of the manual, copy a configuration snippet into a text file, save the file with a name, and copy the file to a directory on your routing platform.

   For example, copy the following snippet to a file and name the file **ex-script-snippet.conf**. Copy the **ex-script-snippet.conf** file to the **/var/tmp** directory on your routing platform.

```
commit {
  file ex-script-snippet.xsl; }
```

2. Move to the hierarchy level that is relevant for this snippet by issuing the following configuration mode command:

```
[edit]
user@host# edit system scripts
[edit system scripts]
```

3. Merge the contents of the file into your routing platform configuration by issuing the **load merge relative** configuration mode command:

```
[edit system scripts]
user@host# load merge relative /var/tmp/ex-script-snippet.conf
load complete
```

For more information about the **load** command, see the *Junos OS CLI User Guide*.

Documentation Conventions

Table 1 on page xxix defines notice icons used in this guide.

Table 1: Notice Icons

Icon	Meaning	Description
	Informational note	Indicates important features or instructions.
	Caution	Indicates a situation that might result in loss of data or hardware damage.
	Warning	Alerts you to the risk of personal injury or death.
	Laser warning	Alerts you to the risk of personal injury from a laser.

Table 2 on page xxix defines the text and syntax conventions used in this guide.

Table 2: Text and Syntax Conventions

Convention	Description	Examples
Bold text like this	Represents text that you type.	To enter configuration mode, type the **configure** command: user@host> **configure**
`Fixed-width text like this`	Represents output that appears on the terminal screen.	`user@host>` **show chassis alarms** `No alarms currently active`
Italic text like this	• Introduces important new terms. • Identifies book names. • Identifies RFC and Internet draft titles.	• A policy *term* is a named structure that defines match conditions and actions. • *Junos System Basics Configuration Guide* • RFC 1997, *BGP Communities Attribute*
Italic text like this	Represents variables (options for which you substitute a value) in commands or configuration statements.	Configure the machine's domain name: [edit] root@# **set system domain-name** *domain-name*
Text like this	Represents names of configuration statements, commands, files, and directories; IP addresses; configuration hierarchy levels; or labels on routing platform components.	• To configure a stub area, include the **stub** statement at the **[edit protocols ospf area area-id]** hierarchy level. • The console port is labeled **CONSOLE**.
< > (angle brackets)	Enclose optional keywords or variables.	**stub** <**default-metric** *metric*>**;**

Table 2: Text and Syntax Conventions *(continued)*

Convention	Description	Examples
| (pipe symbol)	Indicates a choice between the mutually exclusive keywords or variables on either side of the symbol. The set of choices is often enclosed in parentheses for clarity.	**broadcast | multicast** (***string1*** **|** ***string2*** **|** ***string3***)
# (pound sign)	Indicates a comment specified on the same line as the configuration statement to which it applies.	**rsvp { # Required for dynamic MPLS only**
[ ] (square brackets)	Enclose a variable for which you can substitute one or more values.	**community name members [** ***community-ids*** **]**

Indention and braces ({ })	Identify a level in the configuration hierarchy.	[edit]
		routing-options {
		  static {
		  route default {
		  nexthop *address*;
		  retain;
		  }
		  }
		}

; (semicolon)	Identifies a leaf statement at a configuration hierarchy level.	
J-Web GUI Conventions		
Bold text like this	Represents J-Web graphical user interface (GUI) items you click or select.	• In the Logical Interfaces box, select **All Interfaces**. • To cancel the configuration, click **Cancel**.
> (bold right angle bracket)	Separates levels in a hierarchy of J-Web selections.	In the configuration editor hierarchy, select **Protocols>Ospf**.

Documentation Feedback

We encourage you to provide feedback, comments, and suggestions so that we can improve the documentation. You can send your comments to **techpubs-comments@juniper.net**, or fill out the documentation feedback form at https://www.juniper.net/cgi-bin/docbugreport/. If you are using e-mail, be sure to include the following information with your comments:

- Document or topic name
- URL or page number
- Software release version (if applicable)

Requesting Technical Support

Technical product support is available through the Juniper Networks Technical Assistance Center (JTAC). If you are a customer with an active J-Care or JNASC support contract,

or are covered under warranty, and need post-sales technical support, you can access our tools and resources online or open a case with JTAC.

- JTAC policies—For a complete understanding of our JTAC procedures and policies, review the JTAC User Guide located at http://www.juniper.net/us/en/local/pdf/resource-guides/7100059-en.pdf .
- Product warranties—For product warranty information, visit http://www.juniper.net/support/warranty/ .
- JTAC hours of operation—The JTAC centers have resources available 24 hours a day, 7 days a week, 365 days a year.

Self-Help Online Tools and Resources

For quick and easy problem resolution, Juniper Networks has designed an online self-service portal called the Customer Support Center (CSC) that provides you with the following features:

- Find CSC offerings: http://www.juniper.net/customers/support/
- Search for known bugs: http://www2.juniper.net/kb/
- Find product documentation: http://www.juniper.net/techpubs/
- Find solutions and answer questions using our Knowledge Base: http://kb.juniper.net/
- Download the latest versions of software and review release notes: http://www.juniper.net/customers/csc/software/
- Search technical bulletins for relevant hardware and software notifications: https://www.juniper.net/alerts/
- Join and participate in the Juniper Networks Community Forum: http://www.juniper.net/company/communities/
- Open a case online in the CSC Case Management tool: http://www.juniper.net/cm/

To verify service entitlement by product serial number, use our Serial Number Entitlement (SNE) Tool: https://tools.juniper.net/SerialNumberEntitlementSearch/

Opening a Case with JTAC

You can open a case with JTAC on the Web or by telephone.

- Use the Case Management tool in the CSC at http://www.juniper.net/cm/ .
- Call 1-888-314-JTAC (1-888-314-5822 toll-free in the USA, Canada, and Mexico).

For international or direct-dial options in countries without toll-free numbers, see http://www.juniper.net/support/requesting-support.html .

PART 1

Overview

CHAPTER 1

Overview of Configuration and Operations Automation

This chapter contains a brief overview of the configuration and operations automation tools provided by Juniper Networks Junos OS. These tools include commit scripts and macros, operation scripts, and event scripts and event policies.

This chapter discusses the following topics:

Junos Automation Overview

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

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.

Junos automation can capture the knowledge and expertise of experienced network operators and administrators and allow a business to leverage this combined expertise across the organization.

Junos automation scripts can be written in either of two scripting languages: Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX). 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. Scripts written in one language are easily converted to the other.

The following sections describe the different types of functionality for Junos automation:

- Junos Configuration Automation: Commit Scripts on page 4
- Junos Operations Automation: Op Scripts on page 5
- Junos Event Automation: Event Scripts and Event Policy on page 5

Junos Configuration Automation: Commit Scripts

Junos configuration automation uses commit scripts to automate the commit process. Junos OS commit scripts enforce custom configuration rules. When a candidate configuration is committed, it is inspected by each active commit script. If a configuration violates your custom rules, the script can instruct the Junos OS to take appropriate action. A commit script can perform the following actions:

- Generate and display custom warning messages to the user
- Generate and log custom system log (syslog) messages
- Change the configuration to conform to the custom configuration rules
- Generate a commit error and halt the commit operation

Commit scripts, when used in conjunction with macros, allow you to simplify the Junos configuration and, at the same time, extend it with your own custom configuration syntax.

Junos Operations Automation: Op Scripts

Junos operations automation uses op scripts to automate operational tasks and network troubleshooting. Junos OS op scripts can be executed manually in the CLI or upon user login, or they can be called from another script. Op scripts can be constructed to:

- Create custom operational mode commands
- Execute a series of operational mode commands
- Customize the output of operational mode commands
- Shorten troubleshooting time by gathering operational information and iteratively narrowing down the cause of a network problem
- Perform controlled configuration changes

Junos Event Automation: Event Scripts and Event Policy

Junos event automation uses event policy and event scripts to instruct the Junos OS to perform actions in response to system events.

Event Policy

An event policy is an if-then-else construct that defines actions to be executed by the software on receipt of an event such as a system log message or SNMP trap. Event policies can be executed in response to a single system event or to correlated system events. For each policy, you can configure multiple actions including:

- Ignore the event
- Upload a file to a specified destination
- Execute Junos OS operational mode commands
- Execute Junos OS event scripts

Event Scripts

Junos OS event scripts are triggered automatically by defined event policies in response to a system event and can instruct the Junos OS to take immediate action. An event script automates network troubleshooting and network management by doing the following:

- Automatically diagnose and fix problems in the network
- Monitor the overall status of a device
- Run automatically as part of an event policy that detects periodic error conditions
- Change the configuration in response to a problem

Related Documentation

- Commit Script Overview on page 125

CHAPTER 2

Scripts and Event Policy Configuration Statements

This chapter shows the complete configuration statement hierarchy for scripts and for event policy, listing all possible configuration statements and showing their level in the configuration hierarchy. When you are configuring the Junos OS, your current hierarchy level is shown in the banner on the line preceding the **user@host#** prompt.

This chapter is organized as follows:

Any Hierarchy Level

The following statement can be added at any level of the configuration:

```
apply-macro apply-macro-name {
   parameter-name parameter-value;
}
```

[edit event-options] Hierarchy Level

The following statements can be included at the **[edit event-options]** hierarchy level:

```
[edit event-options]
destinations {
    destination-name {
        archive-sites {
            url <password password>;
        }
        transfer-delay seconds;
    }
}
event-scripts {
    file filename {
        checksum (md5 | sha-256 | sha1) hash;
        refresh;
        refresh-from url;
        remote-execution {
            remote-hostname {
                passphrase user-password;
                username user-login;
            }
        }
        source url;
    }
    refresh;
    refresh-from url;
    traceoptions {
        file <filename> <files number> <size size> <world-readable | no-world-readable>;
        flag flag;
        no-remote-trace;
    }
}
generate-event event-name {
    time-interval seconds;
    time-of-day hh:mm:ss;
}
policy policy-name {
    attributes-match {
        event1.attribute-name equals event2.attribute-name;
        event.attribute-name matches regular-expression;
        event1.attribute-name starts-with event2.attribute-name;
    }
    events [ events ];
    within seconds {
        events [ events ];
        not events [ events ];
        trigger (on | after | until) event-count;
    }
    then {
        event-script filename {
            arguments {
                argument-name argument-value;
```

```
        }
        output-filename filename;
        destination destination-name;
      }
      execute-commands {
        commands {
          "command";
        }
        output-filename filename;
        output-format (text | xml);
        destination destination-name;
      }
      ignore;
      raise-trap;
      upload filename (filename | committed) destination destination-name {
        retry-count number retry-interval seconds;
        transfer-delay seconds;
        user-name username;
      }
    }
}
traceoptions {
  file filename <files number> <size size> <world-readable | no-world-readable>;
  flag flag;
}
```

[edit system scripts] Hierarchy Level

The following statements can be configured at the **[edit system]** hierarchy level. This is not a comprehensive list of statements available at the **[edit system]** hierarchy level. For more information about system configuration, see the *Junos OS System Basics Configuration Guide*.

```
[edit system]
scripts {
    commit {
        allow-transients;
        direct-access;
        file filename {
            checksum (md5 | sha-256 | sha1) hash;
            optional;
            refresh;
            refresh-from url;
            source url;
        }
        refresh;
        refresh-from url;
        traceoptions {
            file <filename> <files number> <size size> <world-readable | no-world-readable>;
            flag flag;
            no-remote-trace;
        }
    }
    op {
        file filename {
            arguments {
                argument-name {
                    description descriptive-text;
                }
            }
            checksum (md5 | sha-256 | sha1) hash;
            command filename-alias;
            description descriptive-text;
            refresh;
            refresh-from url;
            source url;
        }
        refresh;
        refresh-from url;
        source url;
        traceoptions {
            file <filename> <files number> <size size> <world-readable | no-world-readable>;
            flag flag;
            no-remote-trace;
        }
    }
}
```

CHAPTER 3

Introduction to the Junos XML API and Junos XML Management Protocol

This chapter discusses the following topics:

XML Overview

Extensible Markup Language (XML) is a language for defining a set of markers, called *tags*, that are applied to a data set or document to describe the function of individual elements and codify the hierarchical relationships between them. Tags look much like Hypertext Markup Language (HTML) tags, but XML is actually a metalanguage used to define tags that best suit the kind of data being marked.

For more details about XML, see *A Technical Introduction to XML* at http://www.xml.com/pub/a/98/10/guide0.html and the additional reference material at the http://www.xml.com site.

The official XML specification from the World Wide Web Consortium (W3C), *Extensible Markup Language (XML) 1.0*, is available at http://www.w3.org/TR/REC-xml .

The following sections discuss general aspects of XML:

Tag Elements

XML has three types of tags: opening tags, closing tags, and empty tags. XML tag names are enclosed in angle brackets and are case sensitive. Items in an XML-compliant document or data set are always enclosed in paired opening and closing tags, and the tags must be properly nested. That is, you must close the tags in the same order in which you opened them. XML is stricter in this respect than HTML, which sometimes uses only opening tags. The following examples show paired opening and closing tags enclosing a value. The closing tags are indicated by the forward slash at the start of the tag name.

```
<interface-state>enabled</interface-state>
<input-bytes>25378</input-bytes>
```

The term *tag element* refers to a three-part set: opening tag, contents, and closing tag. The content can be an alphanumeric character string as in the preceding examples, or can itself be a *container* tag element, which contains other tag elements. For simplicity, the term *tag* is often used interchangeably with *tag element* or *element*.

If a tag element is *empty*—has no contents—it can be represented either as paired opening and closing tags with nothing between them, or as a single tag with a forward slash after the tag name. For example, the notation **<snmp-trap-flag/>** is equivalent to **<snmp-trap-flag></snmp-trap-flag>**.

As the preceding examples show, angle brackets enclose the name of the tag element. This is an XML convention, and the brackets are a required part of the complete tag element name. They are not to be confused with the angle brackets used in the documentation to indicate optional parts of Junos OS CLI command strings.

Junos XML and Junos XML protocol tag elements obey the XML convention that the tag element name indicates the kind of information enclosed by the tags. For example, the name of the Junos XML **<interface-state>** tag element indicates that it contains a description of the current status of an interface on the device, whereas the name of the **<input-bytes>** tag element indicates that its contents specify the number of bytes received.

When discussing tag elements in text, this documentation conventionally uses just the opening tag to represent the complete tag element (opening tag, contents, and closing tag). For example, the documentation refers to the **<input-bytes>** tag to indicate the entire **<input-bytes>*number-of-bytes*</input-bytes>** tag element.

Attributes

XML elements can contain associated properties in the form of *attributes*, which specify additional information about an element. Attributes appear in the opening tag of an element and consist of an attribute name and value pair. The attribute syntax consists of the attribute name followed by an equals sign and then the attribute value enclosed in quotation marks. An XML element can have multiple attributes. Multiple attributes are separated by spaces and can appear in any order.

In the following example, the **configuration** tag element has two attributes, **junos:changed-seconds** and **junos:changed-localtime**.

```
<configuration junos:changed-seconds="1279908006"
junos:changed-localtime="2010-07-23 11:00:06 PDT">
```

The value of the **junos:changed-seconds** attribute is "1279908006", and the value of the **junos:changed-localtime** attribute is "2010-07-23 11:00:06 PDT".

Namespaces

Namespaces allow an XML document to contain the same tag, attribute, or function names for different purposes and avoid name conflicts. For example, many namespaces may define a **print** function, and each may exhibit a different functionality. To use the functionality defined in one specific namespace, you must associate that function with the namespace that defines the desired functionality.

To refer to a tag, attribute, or function from a defined namespace, you must first provide the namespace Uniform Resource Identifier (URI) in your style sheet declaration . You then qualify a tag, attribute, or function from the namespace with the URI. Since a URI is often lengthy, generally a shorter prefix is mapped to the URI.

In the following example the **jcs** prefix is mapped to the namespace identified by the URI http://xml.juniper.net/junos/commit-scripts/1.0, which defines extension functions used in commit, op, and event scripts. The **jcs** prefix is then prepended to the **output** function, which is defined in that namespace.

```
<?xml version=”1.0”?>
 <xsl:stylesheet version=”1.0”
xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  ...
  <xsl:value-of select="jcs:output('The VPN is up.')"/>
</xsl: stylesheet>
```

During processing, the prefix is expanded into the URI reference. Although there may be multiple namespaces that define an **output** element or function, the use of **jcs:output** explicitly defines which **output** function is used. You can choose any prefix to refer to the contents in a namespace, but there must be an existing declaration in the XML document that binds the prefix to the associated URI.

Document Type Definition

An XML-tagged document or data set is *structured*, because a set of rules specifies the ordering and interrelationships of the items in it. The rules define the contexts in which each tagged item can—and in some cases must—occur. A file called a *document type definition*, or *DTD*, lists every tag element that can appear in the document or data set, defines the parent-child relationships between the tags, and specifies other tag characteristics. The same DTD can apply to many XML documents or data sets.

Related Documentation

- Junos XML API and Junos XML Management Protocol Overview on page 15
- XML and the Junos OS on page 13

XML and the Junos OS

Extensible Markup Language (XML) is a standard for representing and communicating information. It is a metalanguage for defining customized tags that are applied to a data set or document to describe the function of individual elements and codify the hierarchical

relationships between them. Junos OS natively supports XML for the operation and configuration of devices running Junos OS.

The Junos OS command-line interface (CLI) and the Junos OS infrastructure communicate using XML. When you issue an operational mode command in the CLI, the CLI converts the command into XML format for processing. After processing, the Junos OS returns the output in the form of an XML document, which the CLI converts back into a readable format for display. Remote client applications also use XML-based data encoding for operational and configuration requests on devices running Junos OS.

The Junos XML API is an XML representation of Junos configuration statements and operational mode commands. It defines an XML equivalent for all statements in the Junos configuration hierarchy and many of the commands that you issue in CLI operational mode. Each operational mode command with a Junos XML counterpart maps to a request tag element and, if necessary, a response tag element.

You can view the XML-formatted output of any operational mode command by issuing the command in the CLI and adding the **| display xml** option. The following example shows the text-fomatted and XML-formatted output for the **show chassis alarms** operational mode command:

```
user@host> show chassis alarms
No alarms currently active

user@host> show chassis alarms | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4B1/junos">
    <alarm-information xmlns="http://xml.juniper.net/junos/10.4B1/junos-alarm">
        <alarm-summary>
            <no-active-alarms/>
        </alarm-summary>
    </alarm-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

You can view the Junos XML API representation of any operational mode command by issuing the command in the CLI and adding the **| display xml rpc** option. The following example shows the Junos XML API tag element for the **show chassis alarms** command.

```
user@host> show chassis alarms | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4B1/junos">
    <rpc>
        <get-alarm-information>
        </get-alarm-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

As shown in the previous example, the **| display xml rpc** option displays the command's corresponding Junos XML API request tag element that is sent to the Junos OS for processing whenever the command is issued. In contrast, the **| display xml** option displays the actual output of the processed command in XML format.

When you issue the **show chassis alarms** operational mode command, the CLI converts the command into its equivalent Junos XML API request tag **<get-alarm-information>** and sends the XML request to the Junos infrastructure for processing. The Junos OS processes the request and returns the **<alarm-information>** response tag element to the CLI. The CLI then converts the XML output into the "No alarms currently active" message that is displayed to the user.

Junos automation scripts use XML to communicate with the host device. The Junos OS provides XML-formatted input to a script. The script processes the input and then returns XML-formatted output to the Junos OS. The script type determines the XML input document that is sent to the script as well as the output document that is returned to the Junos OS for processing. Op script input consists of a blank XML document. Event scripts receive an XML document containing the description of the triggering event. Commit script input consists of an XML representation of the postinheritance candidate configuration file.

Related Documentation

- Junos XML API and Junos XML Management Protocol Overview on page 15
- XML Overview on page 11
- *Junos XML API Configuration Reference*
- *Junos XML API Operational Reference*

Junos XML API and Junos XML Management Protocol Overview

The Junos XML Management Protocol is an XML-based protocol that client applications use to request and change configuration information on routing, switching, and security platforms running Junos OS. It uses an XML-based data encoding for the configuration data and remote procedure calls. The protocol defines basic operations that are equivalent to configuration mode commands in the Junos OS command-line interface (CLI). Applications use the protocol operations to display, edit, and commit configuration statements (among other operations), just as administrators use CLI configuration mode commands such as **show**, **set**, and **commit** to perform those operations.

The Junos XML API is an XML representation of Junos configuration statements and operational mode commands. Junos XML configuration tag elements are the content to which the Junos XML protocol operations apply. Junos XML operational tag elements are equivalent in function to operational mode commands in the CLI, which administrators use to retrieve status information for a device.

Client applications request or change information on a device by encoding the request with tag elements from the Junos XML management protocol and Junos XML API and sending it to the Junos XML protocol server on the device. The Junos XML protocol server is integrated into the Junos OS and does not appear as a separate entry in process listings. The Junos XML protocol server directs the request to the appropriate software modules within the device, encodes the response in Junos XML and Junos XML protocol tag elements, and returns the result to the client application. For example, to request information about the status of a device's interfaces, a client application sends the Junos XML API **<get-interface-information>** request tag element . The Junos XML protocol server

gathers the information from the interface process and returns it in the Junos XML API **<interface-information>** response tag element.

You can use the Junos XML management protocol and Junos XML API to configure devices running Junos OS or request information about the device configuration or operation. You can write client applications to interact with the Junos XML protocol server, and you can also utilize the Junos XML protocol to build custom end-user interfaces for configuration and information retrieval and display, such as a Web browser-based interface.

Related Documentation

- Advantages of Using the Junos XML Management Protocol and Junos XML API on page 16
- XML and the Junos OS on page 13
- XML Overview on page 11

Advantages of Using the Junos XML Management Protocol and Junos XML API

The Junos XML management protocol and Junos XML API fully document all options for every supported Junos operational request, all statements in the Junos configuration hierarchy, and basic operations that are equivalent to configuration mode commands. The tag names clearly indicate the function of an element in an operational or configuration request or a configuration statement.

The combination of meaningful tag names and the structural rules in a DTD makes it easy to understand the content and structure of an XML-tagged data set or document. Junos XML and Junos XML protocol tag elements make it straightforward for client applications that request information from a device to parse the output and find specific information.

Parsing Device Output

The following example illustrates how the Junos XML API makes it easier to parse device output and extract the needed information. The example compares formatted ASCII and XML-tagged versions of output from a device running Junos OS.

The formatted ASCII follows:

```
Physical interface: fxp0, Enabled, Physical link is Up
  Interface index: 4, SNMP ifIndex: 3
```

The corresponding XML-tagged version is:

```
<interface>
  <name>fxp0</name>
  <admin-status>enabled</admin-status>
  <operational-status>up</operational-status>
  <index>4</index>
  <snmp-index>3</snmp-index>
</interface>
```

When a client application needs to extract a specific value from formatted ASCII output, it must rely on the value's location, expressed either absolutely or with respect to labels

or values in adjacent fields. Suppose that the client application wants to extract the interface index. It can use a regular-expression matching utility to locate specific strings, but one difficulty is that the number of digits in the interface index is not necessarily predictable. The client application cannot simply read a certain number of characters after the **Interface index:** label, but must instead extract everything between the label and the subsequent label **SNMP ifIndex** and also account for the included comma.

A problem arises if the format or ordering of text output changes in a later version of the Junos OS. For example, if a **Logical index** field is added following the interface index number, the new formatted ASCII might appear as follows:

```
Physical interface: fxp0, Enabled, Physical link is Up
  Interface index: 4, Logical index: 12, SNMP ifIndex: 3
```

An application that extracts the interface index number delimited by the **Interface index:** and **SNMP ifIndex:** labels now obtains an incorrect result. The application must be updated manually to search for the **Logical index:** label as the new delimiter.

In contrast, the structured nature of XML-tagged output enables a client application to retrieve the interface index by extracting everything within the opening **<index>** tag and closing **</index>** tag. The application does not have to rely on an element's position in the output string, so the Junos XML protocol server can emit the child tag elements in any order within the **<interface>** tag element. Adding a new **<logical-index>** tag element in a future release does not affect an application's ability to locate the **<index>** tag element and extract its contents.

Displaying Device Output

XML-tagged output is also easier to transform into different display formats than formatted ASCII output. For instance, you might want to display different amounts of detail about a given device component at different times. When a device returns formatted ASCII output, you have to write special routines and data structures in your display program to extract and show the appropriate information for a given detail level. In contrast, the inherent structure of XML output is an ideal basis for a display program's own structures. It is also easy to use the same extraction routine for several levels of detail, simply ignoring the tag elements you do not need when creating a less detailed display.

Related Documentation

- Junos XML API and Junos XML Management Protocol Overview on page 15
- XML Overview on page 11

CHAPTER 4

Understanding XSLT

This chapter contains some overview material, intended as a brief introduction to Extensible Stylesheet Language Transformations (XSLT) and XML Path Language (XPath). It is not a comprehensive user guide for XSLT or XPath. This chapter discusses the following topics:

XSLT Overview

Commit scripts, op scripts, and event scripts can be written in Extensible Stylesheet Language Transformations (XSLT), which is a standard for processing Extensible Markup Language (XML) data. XSLT is developed by the World Wide Web Consortium (W3C) and is accessible at http://www.w3c.org/TR/xslt .

XSLT Advantages

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 (such as additions or deletions) to building a completely new document hierarchy.

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 (which the Junos OS communicates internally as XML). 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.

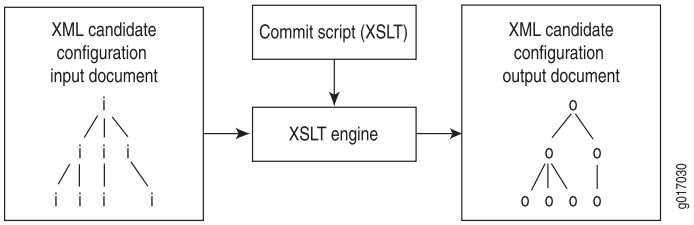
Although XSLT provides a powerful scripting ability, its focus is specific and limited. It does not make the Junos OS vulnerable to arbitrary or malicious programmers. XSLT restricts programmers from performing haphazard operations, such as opening random Transmission Control Protocol (TCP) ports, forking numerous processes, or sending e-mail. The only action available in XSLT is to generate XML, and the XML is interpreted by the UI according to fixed semantics. An XSLT script can output only XML data, which is directly processed by the UI infrastructure to allow only the specific abilities listed above—generating error, warning, and system log messages, and persistent and transient configuration changes. This means that the impact of commit scripts, op scripts, and event scripts on the device is well-defined and can be viewed inside the command-line interface (CLI), using commands added for that purpose.

XSLT Engine

XSLT is a language for transforming one XML document into another XML document. The basic model 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. For commit scripts, op scripts, and event scripts, the XSLT engine is a function of the Junos OS management process (mgd).

Figure 1 on page 20 shows the relationship between an XSLT commit script and the XSLT engine.

Figure 1: Flow of XSLT Commit Script Through the XSLT Engine



XSLT Concepts

XSLT has seven basic concepts. These are summarized in Table 3 on page 20.

Table 3: XSLT Concepts

XSLT Concepts	Description
XPath	Expression syntax for specifying a node in the input document
Templates	Mechanism for mapping input hierarchies to instructions that handle them
Parameters	Mechanism for passing arguments to templates

Table 3: XSLT Concepts *(continued)*

XSLT Concepts	Description
Variables	Mechanism for defining read-only references to nodes
Programming instructions	Mechanism for defining logic in XSLT
Recursion	Mechanism by which templates call themselves to facilitate looping
Context (Dot)	Node currently being inspected in the input document

Related Documentation

- XPath Overview on page 21
- XSLT Context (Dot) Overview on page 33
- XSLT Parameters Overview on page 26
- XSLT Programming Instructions Overview on page 30
- XSLT Recursion Overview on page 32
- XSLT Templates Overview on page 24
- XSLT Variables Overview on page 29

XSLT Namespace

The XSLT namespace has the Uniform Resource Identifier (URI) http://www.w3.org/1999/XSL/Transform. The namespace must be included in the stylesheet declaration of a script in order for the XSLT processor to recognize and use XSLT elements and attributes. The following example declares the XSLT namespace and associates the **xsl** prefix with the URI.

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="route">
    ...
    </xsl:template>
</xsl:stylesheet
```

Once the XSLT namespace is declared in a script, you use elements and attributes from the namespace by adding the associated prefix, which in this case is **xsl**, to the tag or attribute name. In the preceding example, the XSLT processor knows to treat **xsl:template** as an XSLT instruction. During processing, the **xsl** prefix is expanded into the URI reference, and the functionality of the **template** element is defined by the XSLT namespace. For more information about namespaces, see “XML Overview” on page 11.

XPath Overview

XSLT uses the XML Path Language (XPath) standard to specify and locate elements in the input document’s XML hierarchy. XPath’s powerful expression syntax enables you to define complex criteria for selecting portions of the XML input document.

Nodes and Axes

XPath views every piece of the document hierarchy as a *node*. For commit scripts, op scripts, and event scripts, 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:

- **<host-name>my-router</host-name>**—Element node
- **my-router**—Text node
- **inactive="inactive"**—Attribute node

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 the descendents 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.

NOTE: In Junos OS commit scripts, the context node concept corresponds to Junos OS hierarchy levels. For example, the /configuration/system/domain-name XPath expression sets the context node to the [edit system domain-name] hierarchy level.

We recommend including the <xsl:template match="configuration"> template in all commit scripts. This element allows you to exclude the /configuration/ root element from all XPath expressions in programming instructions (such as <xsl:for-each> or <xsl:if>) in the script, thus allowing you to begin XPath expressions at a Junos hierarchy level (for example, system/domain-name). For more information, see "Required Boilerplate for Commit Scripts" on page 133.

Path and Predicate Syntax

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 (**..**).

The 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.

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 always have to 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, see “Summary of XPath and XSLT Constructs” on page 95. We also recommend consulting a comprehensive XPath reference guide. XPath is fully described in the W3C specification at **http://w3c.org/TR/xpath**.

XSLT Templates Overview

An XSLT script consists of one or more sets of rules called *templates*. Each template is a segment of code that contains rules to apply when a specified node is matched. You use the **<xsl:template>** element to build templates.

There are two types of templates, named and unnamed (or match), and they are described in the following sections.

- Unnamed (Match) Templates on page 24
- Named Templates on page 25

Unnamed (Match) Templates

Unnamed templates, also known as match templates, include a **match** attribute that contains an XPath expression to specify the criteria for nodes upon which the template should be invoked. In the following example, the template applies to the 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.**.

```
<xsl:template match="route[starts-with(next-hop, '10.10.')]">
  <!-- ... body of the template ... -->
</xsl:template>
```

By default, when XSLT processes a document, it recursively traverses the entire document hierarchy, inspecting each node, looking for a template that matches the current node. When a matching template is found, the contents of that template are evaluated.

The **<xsl:apply-templates>** element can be used inside an unnamed template to limit and control XSLT's default, hierarchical traversal of nodes. If the **<xsl:apply-templates>** element has a **select** attribute, only nodes matching the XPath expression defined by the attribute are traversed. Otherwise all children of the context node are traversed. If the **select** attribute is included, but does not match any nodes, nothing is traversed and nothing happens.

In the following example, the template rule matches the **<route>** element in the XML hierarchy. All the nodes containing a **changed** attribute are processed. All **<route>** elements containing a **changed** attribute are replaced with a **<new>** element.

```
<xsl:template match="route">
  <new>
    <xsl:apply-templates select="*[@changed]"/>
  </new>
</xsl:template>
```

Using unnamed templates allows the script to ignore the location of a tag in the XML hierarchy. For example, if you want to convert all **<author>** tags into ** ** tags, using templates enables you to write a single rule that converts all **<author>** tags, regardless of their location in the input XML document.

For more information about how unnamed templates are used in scripts, see **xsl:template match="/" Template**.

Named Templates

Named templates operate like functions in traditional programming languages, although with a verbose syntax. When the complexity of a script increases or a code segment appears in multiple places, you can modularize the code and create named templates. Like functions, named templates accept arguments and run only when explicitly called.

You create a named template by using the **<xsl:template>** element and defining the **name** attribute, which is similar to a function name in traditional programming languages. Use the **<xsl:param>** tag and its **name** attribute to define parameters for the named template, and optionally include the **select** attribute to declare default values for each parameter. The **select** attribute can contain XPath expressions. If the **select** attribute is not defined, the parameter defaults to an empty string.

The following example creates a template named **my-template** and defines three parameters, one of which defaults to the string **false**, and one of which defaults to the contents of the element node named **name** that is a child of the current context node. If the script calls the template and does not pass in a parameter, the default value is used.

```
<xsl:template name="my-template">
  <xsl:param name="a"/>
  <xsl:param name="b" select="'false'"/>
  <xsl:param name="c" select="name"/>
   <!-- ... body of the template ... -->
</xsl:template>
```

To invoke a named template in a script, use the **<xsl:call-template>** element. The **name** attribute is required and defines the name of the template being called. When processed, the **<xsl:call-template>** element is replaced by the contents of the **<xsl:template>** element it names.

When you invoke a named template, you can pass arguments into the template by including the **<xsl:with-param>** child element and specifying the **name** attribute. The value of the **<xsl:with-param> name** attribute must match a parameter defined in the actual template; otherwise the parameter is ignored. Optionally, you can set a value for each parameter with either the **select** attribute or the content of the **<xsl:with-param>** element. If you do not define a value for the parameter in the calling environment, the script passes in the current value of the parameter if it was previously initialized, or it generates an error if the parameter was never declared. For more information about passing parameters, see “XSLT Parameters Overview” on page 26.

In the following example, the template **my-template** is called with the parameter **c** containing the contents of the element node named **other-name** that is a child of the current context node.

```
<xsl:call-template name="my-template">
  <xsl:with-param name="c" select="other-name"/>
</xsl:call-template>
```

For an example showing how to use named templates in a commit script, see “Example: Requiring and Restricting Configuration Statements” on page 201.

Related Documentation

- XSLT Parameters Overview on page 26
- **xsl:apply-templates on page 99**
- **xsl:call-template on page 100**
- **xsl:param on page 104**
- **xsl:template on page 105**
- **xsl:template match="/" Template on page 90**
- **xsl:with-param on page 108**

XSLT Parameters Overview

Parameters can be passed to either named or unnamed templates. Inside the template, parameters must be declared and can then be referenced by prefixing their name with the dollar sign ($).

Declaring Parameters

The scope of a parameter can be global or local. A parameter whose value is set by Junos at script initialization must be defined as a global parameter. Global parameter declarations are placed just after the stylesheet declarations. A script can assign a default value to the global parameter, which is used in the event that Junos does not give a value to the parameter.

```
<?xml version="1.0" standalone="yes"?>
  <xsl stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:junos="http://xml.juniper.net/junos/*/junos"
     xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
     xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0"
     xmlns:ext="http://xmlsoft.org/XSLT/namespace" version="1.0">

<!-- global parameter -->
<xsl:param name="interface1"/>
```

Local parameters must be declared at the beginning of a block and their scope is limited to the block in which they are declared. Inside a template, you declare parameters using the **<xsl:param>** tag and **name** attribute. Optionally, declare default values for each parameter by including the **select** attribute, which can contain XPath expressions. If a template is invoked without the parameter, the default expression is evaluated, and the results are assigned to the parameter. If you do not define a default value in the template, the parameter defaults to an empty string.

The following named template **print-host-name** declares the parameter **message** and defines a default value:

```
<xsl:template name="print-host-name">
   <xsl:param name="message"
         select="concat('host-name: ', system/host-name)"/>
   <xsl:value-of select="$message"/>
</xsl:template>
```

The template accesses the value of the **message** parameter by prefixing the parameter name with the dollar sign ($).

Passing Parameters

When you invoke a template, you pass arguments into the template using the **<xsl:with-param>** element and **name** attribute. The value of the **<xsl:with-param> name** attribute must match the name of a parameter defined in the actual template; otherwise the parameter is ignored. Optionally, for each parameter you pass to a template, you can define a value using either the **select** attribute or the contents of the **<xsl:with-param>** element.

The parameter value that gets used in a template depends on how the template is called. The following three examples, which call the **print-host-name** template, illustrate the possible calling environments.

If you call a template but do not include the **<xsl:with-param>** element for a specific parameter, the default expression defined in the template is evaluated, and the results are assigned to the parameter. If there is no default value for that parameter in the template, the parameter defaults to an empty string. The following example calls the named template **print-host-name** but does not include any parameters in the call. In this case, the named template will use the default value for the **message** parameter that was defined in the **print-host-name** template, or an empty string if no default exists.

```
<xsl:template match="configuration">
    <xsl:call-template name="print-host-name"/>
</xsl:template>
```

If you call a template and include a parameter, but do not define a value for the parameter in the calling environment, the script passes in the current value of the parameter if it was previously initialized, or it generates an error if the parameter was never declared. The following example calls the named template **print-host-name** and passes in the **message** parameter, but does not include a value. If **message** is declared and initialized in the script, and the scope is visible to the block, the current value of **message** is used. If **message** is declared in the script but not initialized, the value of **message** will be an empty string. If **message** has not been declared, the script produces an error.

```
<xsl:template match="configuration">
    <xsl:call-template name="print-host-name">
        <xsl:with-param name="message"/>
    </xsl:call-template>
</xsl:template>
```

If you call a template, include the parameter, and define a value for the parameter, the template uses the provided value. The following example calls the named template **print-host-name** with the **message** parameter and a defined value, so the template uses the new value.

```
<xsl:template match="configuration">
    <xsl:call-template name="print-host-name">
        <xsl:with-param name="message"
            select=concat"('Host-name passed in: ', system/host-name)"/>
    </xsl:call-template>
</xsl:template>
```

Example: Parameters and Match Templates

The following template matches on /, the root of the XML document. It then generates an element named **<outside>**, which is added to the output document, and instructs the Junos OS management process (mgd) to recursively apply templates to the **configuration/system** subtree. The parameter **host** is used in the processing of any matching nodes.The value of the **host** parameter is the value of the **host-name** statement at the [**edit system**] level of the configuration hierarchy.

```
<xsl:template match="/">
  <outside>
    <xsl:apply-templates select="configuration/system">
        <xsl:with-param name="host" select="configuration/system/host-name"/>
    </xsl:apply-templates>
  </outside>
</xsl:template>
```

The following template matches the **<system>** element, which is the top of the subtree selected in the previous example. The **host** parameter is declared with no default value. An **<inside>** element is generated, which contains the value of the **host** parameter that was defined in the **<xsl:with-param>** tag in the previous example.

```
<xsl:template match="system">
  <xsl:param name="host"/>
  <inside>
    <xsl:value-of select="$host"/>
  </inside>
</xsl:template>
```

Example: Parameters and Named Templates

The following named template **report-changed** declares two parameters: **dot**, which defaults to the current node, and **changed**, which defaults to the **changed** attribute of the node **$dot**.

```
<xsl:template name="report-changed">
  <xsl:param name="dot" select="."/>
  <xsl:param name="changed" select="$dot/@changed"/>
  <!-- ... -->
</xsl:template>
```

The next stanza calls the **report-changed** template and defines a source for the **changed** attribute different from the default source defined in the **report-changed** template. When the **report-changed** template is invoked, it will use the newly defined source for the **changed** attribute in place of the default source.

```
<xsl:template match="system">
  <xsl:call-template name="report-changed">
    <xsl:with-param name="changed" select="../@changed"/>
  </xsl:call-template>
</xsl:template>
```

Likewise, the template call can include the **dot** parameter and define a source other than the default current node, as shown here:

```
<xsl:template match="system">
  <xsl:call-template name="report-changed">
    <xsl:with-param name="dot" select="../../>
```

```
    </xsl:call-template>
  </xsl:template>
```

Related Documentation

- XSLT Templates Overview on page 24
- **xsl:param on page 104**
- **xsl:with-param on page 108**

XSLT Variables Overview

You declare variables using the **<xsl:variable>** element. The **name** attribute specifies the name of the variable, which is case-sensitive. Once you declare a variable, you can reference it within an XPath expression using the variable name prefixed with a dollar sign ($).

Variables are immutable; you can set the value of a variable only when you declare the variable, after which point, the value is fixed. You initialize a variable by including the **select** attribute and an expression in the **<xsl:variable>** tag. The following example declares and initializes the variable **location**. The **location** variable is then used to initialize the **message** variable.

```
<xsl:variable name="location" select="$dot/@location"/>
<xsl:variable name="message" select="concat('We are in ', $location, ' now.')"/>
```

You can define both local and global variables. Variables are global if they are children of the **<xsl:stylesheet>** element. Otherwise, they are local. The value of a global variable is accessible anywhere in the stylesheet. The scope of a local variable is limited to the template or code block in which it is defined.

XSLT variables can store any values that you can calculate or statically define. This includes data structures, XML hierarchies, and combinations of text and parameters. For example, you could assign the XML output of an operational mode command to a variable and then access the hierarchy within the variable.

The following template declares the **message** variable. The **message** variable includes both text and parameter values. The template generates a system log message by referring to the value of the message variable. The resulting system log message is as follows:

Device *device-name* was changed on *date* by user '*user*.'

```
<xsl:template name="emit-syslog">
  <xsl:param name="user"/>
  <xsl:param name="date"/>
  <xsl:param name="device"/>
  <xsl:variable name="message">
    <xsl:text>Device </xsl:text>
    <xsl:value-of select="$device"/>
    <xsl:text> was changed on </xsl:text>
    <xsl:value-of select="$date"/>
    <xsl:text> by user '</xsl:text>
    <xsl:value-of select="$user"/>
    <xsl:text>.'</xsl:text>
  </xsl:variable>
```

```
    <syslog>
      <message>
        <xsl:value-of select="$message"/>
      </message>
    </syslog>
  </xsl:template>
```

Table 4 on page 30 provides examples of XSLT variable declarations along with pseudocode explanations.

Table 4: Examples and Pseudocode for XSLT Variable Declaration

Variable Declaration	Pseudocode Explanation
<xsl:variable name="mpls" select="protocols/mpls"/>	Assigns the **[edit protocols mpls]** hierarchy level to the variable named **mpls**.
<xsl:variable name="color" select="data[name = 'color']/value"/>	Assigns the value of the **color** macro parameter to a variable named **color**. The **<data>** element in the XPath expression is useful in commit script macros. For more information, see "Creating a Macro to Read the Custom Syntax and Generate Related Configuration Statements" on page 177.

XSLT Programming Instructions Overview

XSLT has a number of traditional programming instructions. Their form tends to be verbose, because their syntax is built from XML elements.

The XSLT programming instructions most commonly used in commit, op, and event scripts, which provide flow control within a script, are described in the following sections:

- <xsl:choose> Programming Instruction on page 30
- <xsl:for-each> Programming Instruction on page 31
- <xsl:if> Programming Instruction on page 31
- Sample XSLT Programming Instructions and Pseudocode on page 31

<xsl:choose> Programming Instruction

The **<xsl:choose>** instruction is a conditional construct that causes different instructions to be processed in different circumstances. It is similar to a switch statement in traditional programming languages. The **<xsl:choose>** instruction contains one or more **<xsl:when>** elements, each of which tests an XPath expression. If the test evaluates to true, the XSLT processor executes the instructions in the **<xsl:when>** element. After the XSLT processor finds an XPath expression in an **<xsl:when>** element that evaluates to true, the XSLT processor ignores all subsequent **<xsl:when>** elements contained in the **<xsl:choose>** instruction, even if their XPath expressions evaluate to true. In other words, the XSLT processor processes only the instructions contained in the first **<xsl:when>** element whose **test** attribute evaluates to true. If none of the **<xsl:when>** elements' **test** attributes evaluate to true, the content of the optional **<xsl:otherwise>** element, if there is one, is processed.

The **<xsl:choose>** instruction is similar to a switch statement in other programming languages. The **<xsl:when>** element is the “case” of the switch statement, and you can add any number of **<xsl:when>** elements. The **<xsl:otherwise>** element is the “default” of the switch statement.

```
<xsl:choose>
  <xsl:when test="xpath-expression">
    ...
  </xsl:when>
  <xsl:when test="another-xpath-expression">
    ...
  </xsl:when>
  <xsl:otherwise>
    ...
  </xsl:otherwise>
</xsl:choose>
```

<xsl:for-each> Programming Instruction

The **<xsl:for-each>** element tells the XSLT processor to gather together a set of nodes and process them one by one. The nodes are selected by the XPath expression specified by the **select** attribute. Each of the nodes is then processed according to the instructions held in the **<xsl:for-each>** construct.

```
<xsl:for-each select="xpath-expression">
  ...
</xsl:for-each>
```

Code inside the **<xsl:for-each>** instruction is evaluated recursively for each node that matches the XPath expression. That is, the current context is moved to each node selected by the **<xsl:for-each>** clause, and processing is relative to that current context.

In the following example, the **<xsl:for-each>** construct recursively processes each node in the **[system syslog file]** hierarchy. It updates the current context to each matching node and prints the value of the **name** element, if one exists, that is a child of the current context.

```
<xsl:for-each select="system/syslog/file">
  <xsl:value-of select=”name”/>
</xsl:for-each>
```

<xsl:if> Programming Instruction

An **<xsl:if>** programming instruction is a conditional construct that causes instructions to be processed if the XPath expression held in the **test** attribute evaluates to **true**.

```
<xsl:if test="xpath-expression">
  ...executed if test expression evaluates to true
</xsl:if>
```

There is no corresponding else clause.

Sample XSLT Programming Instructions and Pseudocode

Table 5 on page 32 presents examples that use several XSLT programming instructions along with pseudocode explanations.

Table 5: Examples and Pseudocode for XSLT Programming Instructions

Programming Instruction	Pseudocode Explanation
<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**.
<xsl:for-each select="interfaces/interface[starts-with(name, 'ge-')]/unit">	For each Gigabit Ethernet interface configured at the **[edit interfaces ge-*fpc/pic/port* unit *logical-unit-number*]** hierarchy level.

<xsl:for-each select="data[not(value)]/name">	Select any macro parameter that does not contain a parameter value.
	
	In other words, match all **apply-macro** statements of the following form:
	
	apply-macro *apply-macro-name* {
	parameter-name;
	}
	
	And ignore all **apply-macro** statements of the form:
	
	apply-macro *apply-macro-name* {
	parameter-name parameter-value;
	}

<xsl:if test="not(system/host-name)">	If the **host-name** statement is not included at the **[edit system]** hierarchy level.
<xsl:if test="apply-macro[name = 'no-igp']	If the **apply-macro** statement named **no-igp** is included at the current hierarchy level.
<xsl:if test="not(../apply-macro[name = 'no-ldp'])	If the **apply-macro** statement with the name **no-ldp** is not included two hierarchy levels above the current hierarchy level.

XSLT Recursion Overview

XSLT depends on recursion as a looping mechanism. Recursion occurs when a section of code calls itself, either directly or indirectly. Both named and unnamed templates can use recursion, and different templates can use mutual recursion, one calling another that in turn calls the first.

To avoid infinite recursion and excessive consumption of system resources, the Junos OS management process (mgd) limits the maximum recursion to 5000 levels. If this limit is reached, the script fails.

In the following example, an unnamed template matches on a **<count>** element. It then calls the **<count-to-max>** template, passing the value of the **count** element as **max**. The **<count-to-max>** template starts by declaring both the **max** and **cur** parameters and setting the default value of each to **1** (one). Although the optional default value for **max** is one, the template will use the value passed in from the **count** template. Then the current value of **$cur** is emitted in an **<out>** element. Finally, if **$cur** is less than **$max**, the **<count-to-max>** template recursively invokes itself, passing **$cur + 1** as **cur**. This recursive pass then outputs the next number and repeats the recursion until **$cur** equals **$max**.

```
<xsl:template match="count">
  <xsl:call-template name="count-to-max">
    <xsl:with-param name="max" select="."/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="count-to-max">
  <xsl:param name="cur" select="'1'"/>
  <xsl:param name="max" select="'1'"/>

  <out><xsl:value-of select="$cur"/></out>

  <xsl:if test="$cur &lt; $max">
    <xsl:call-template name="count-to-max">
      <xsl:with-param name="cur" select="$cur + 1"/>
      <xsl:with-param name="max" select="$max"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>
```

Given a **max** value of **10**, the values contained in the **<out>** tag are **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, and **10**.

XSLT Context (Dot) Overview

The current context node changes as an **<xsl:apply-templates>** instruction traverses the document hierarchy and as an **<xsl:for-each>** instruction examines each node that matches an XPath expression. All relative node references are relative to the current context node. This node is abbreviated " **.**" (read: dot) and can be referred to in XPath expressions, allowing explicit references to the current node.

The following example contains four uses for "**.**" . The **system** node is saved in the **system** variable for use inside the **<xsl:for-each>** instruction, where the value of "**.**" will have changed. The **for-each select** expression uses "**.**" to mean the value of the **name** element. The "**.**" is then used to pull the value of the **name** element into the **<tag>** element. The **<xsl:if>** test then uses "**.**" to reference the value of the current context node.

```
<xsl:template match="system">
  <xsl:variable name="system" select="."/>
  <xsl:for-each select="name-server/name[starts-with(., '10.')]">
    <tag><xsl:value-of select="."/></tag>
    <xsl:if test=". = '10.1.1.1'">
```

```
        <match>
          <xsl:value-of select="$system/host-name"/>
        </match>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
```

CHAPTER 5

Understanding SLAX

This chapter discusses the following topics:

SLAX Overview

Stylesheet Language Alternative Syntax (SLAX) is a language for writing Junos OS commit scripts, op scripts, and event scripts. It is an alternative to Extensible Stylesheet Language Transformations (XSLT). SLAX has a distinct syntax similar to that of C and Perl, but the same semantics as XSLT.

SLAX Advantages

XSLT is a powerful and effective tool for handling Extensible Markup Language (XML) that works well for machine-to-machine communication, but its XML-based syntax is inconvenient for the development of complex programs.

SLAX has a simple syntax that follows the style of C and PERL. It provides a practical and succinct way to code, thus allowing you to create readable, maintainable commit, op, and event scripts. SLAX removes XPath expressions and programming instructions from XML elements. XML angle brackets and quotation marks are replaced by parentheses and curly brackets (**{ }**), which are the familiar delimiters of C and PERL.

The benefits of SLAX are particularly strong for programmers who are not already accustomed to XSLT, because SLAX allows them to concentrate on the new programming

topics introduced by XSLT, rather than concentrating on learning a new syntax. For example, SLAX allows 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
- Define named templates with a syntax resembling a function definition
- Invoke named templates with a syntax resembling a function call
- Simplify namespace declarations
- Reduce the clutter in your scripts
- Write more readable scripts

For examples of commit and op scripts written in SLAX, see "Commit Script Examples" on page 201 and "Op Script Examples" on page 329.

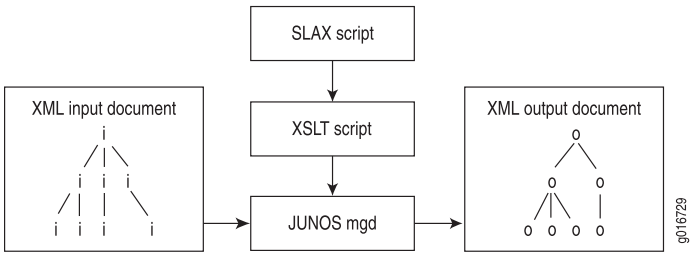
How SLAX Works

SLAX functions as a preprocessor for XSLT. The Junos OS internally translates SLAX programming instructions (such as **if** and **else** statements) into the equivalent XSLT instructions (such as **<xsl:if>** and **<xsl:choose>** elements). After this translation, the XSLT transformation engine—which, for the Junos OS, is the Junos OS management (mgd) process—is invoked.

SLAX does not affect the expressiveness of XSLT; it only makes XSLT easier to use. The underlying SLAX constructs are completely native to XSLT. SLAX adds nothing to the XSLT engine. The SLAX parser parses an input document and builds an XML tree identical to the one produced when the XML parser reads an XSLT document.

Figure 2 on page 36 shows the flow of SLAX script input and output.

Figure 2: SLAX Script Input and Output



Related Documentation

- Converting Scripts Between SLAX and XSLT on page 37
- SLAX Elements and Element Attributes Overview on page 41

- SLAX Statements Overview on page 51
- SLAX Syntax Rules Overview on page 39
- SLAX Templates Overview on page 43
- SLAX Variables Overview on page 50
- XPATH Expressions Overview for SLAX on page 42
- XSLT Overview on page 19

Converting Scripts Between SLAX and XSLT

SLAX is a C-like alternative syntax to XSLT and can be viewed as a preprocessor for XSLT. Before the Junos OS invokes the XSLT processor, the software converts any SLAX constructs in the script (such as **if/then/else**) to equivalent XSLT constructs (such as **<xsl:choose>** and **<xsl:if>**). For more information about SLAX, see "SLAX Overview" on page 35.

You can use the **request system scripts convert** operational mode command to convert a script written in SLAX or XSLT into the alternate language. If you have existing XSLT scripts, conversion to SLAX allows C and PERL programmers to more easily read and maintain the scripts. In addition, converting a script and studying the results facilitates learning the differences between the two languages.

The following sections explain how to convert a script from one language to the other:

- Converting a Script from SLAX to XSLT on page 37
- Converting a Script from XSLT to SLAX on page 38

Converting a Script from SLAX to XSLT

To convert a SLAX script to XSLT, issue the **request system scripts convert slax-to-xslt** operational mode command, and specify the source file, the destination directory, and, optionally, a destination file. The source script is the basis for the new script. The source script is not overwritten by the new script.

The command syntax is:

user@host> **request system scripts convert slax-to-xslt source *source* destination *dest***

The following three examples show the command using a source and destination directory relevant to the default storage location for the type of script being converted:

```
user@host> request system scripts convert slax-to-xslt source /var/db/scripts/op/script1.slax
destination /var/db/scripts/op/script1.xsl
conversion complete
```

```
user@host> request system scripts convert slax-to-xslt source /var/db/scripts/event/script1.slax
destination /var/db/scripts/event/script1.xsl
conversion complete
```

```
user@host> request system scripts convert slax-to-xslt source
/var/db/scripts/commit/script1.slax destination /var/db/scripts/commit/script1.xsl
```

```
conversion complete
```

When you issue the **slax-to-xslt** conversion command, the script1.slax file remains unchanged in the source directory and a new script called script1.xsl is added to the destination directory.

```
user@host> file list /var/db/scripts/op
script1.slax
script1.xsl
```

If you specify only the destination directory and do not specify a destination filename, the generated file is named SLAX-Conversion-Temp.*xxxxx* where *xxxxx* is a randomly generated series of characters.

```
user@host> request system scripts convert slax-to-xslt source /var/db/scripts/op/script1.slax
destination /var/db/scripts/op/
conversion complete

user@host> file list /var/db/scripts/op
SLAX-Conversion-Temp.SlhIr
script1.slax
```

Converting a Script from XSLT to SLAX

To convert an XSLT script to SLAX, issue the **request system scripts convert xslt-to-slax** operational mode command, and specify the source file, the destination directory, and, optionally, a destination file. The source script is the basis for the new script. The source script is not overwritten by the new script.

The command syntax is:

user@host> **request system scripts convert xslt-to-slax source** *source* **destination** *dest*

The following three examples show the command using a source and destination directory relevant to the default storage location for the type of script being converted:

```
user@host> request system scripts convert xslt-to-slax source /var/db/scripts/op/script1.xsl
destination /var/db/scripts/op/script1.slax
conversion complete

user@host> request system scripts convert xslt-to-slax source /var/db/scripts/event/script1.xsl
destination /var/db/scripts/event/script1.slax
conversion complete

user@host> request system scripts convert xslt-to-slax source /var/db/scripts/commit/script1.xsl
destination /var/db/scripts/commit/script1.slax
conversion complete
```

When you issue the **xslt-to-slax** conversion command, the script1.xsl file remains unchanged in the source directory, and a new script called script1.slax is added to the destination directory.

```
user@host> file list /var/db/scripts/op
script1.slax
script1.xsl
```

If you specify only the destination directory and do not specify a destination filename, the generated file is named SLAX-Conversion-Temp.*xxxxx* where *xxxxx* is a randomly generated series of characters.

```
user@host> request system scripts convert xslt-to-slax source /var/db/scripts/op/script1.xsl
destination /var/db/scripts/op/
conversion complete

user@host> file list /var/db/scripts/op
SLAX-Conversion-Temp.Vosnd
script1.xsl
```

Related Documentation

- SLAX Overview on page 35

SLAX Syntax Rules Overview

SLAX syntax rules are similar to those of traditional programming languages like C and PERL. The following sections discuss general aspects of SLAX syntax rules:

- Code Blocks on page 39
- Comments on page 39
- Line Termination on page 40
- Strings on page 40

Code Blocks

SLAX delimits blocks of code with curly braces. Code blocks, which may define the boundaries of an element, a hierarchy, or a segment of code, can be at the same level as or nested within other code blocks. Declarations defined within a particular code block have a scope that is limited to that block.

The following example shows two blocks of code. Curly braces define the bounds of the **match /** block. The second block, containing the **<op-script-results>** element, is nested within the first.

```
match / {
  <op-script-results> {
      <output> "Script summary:";
  }
}
```

Comments

In SLAX, you can add comments anywhere in a script. Commenting a script increases readability for all users, including the author, who may need to return to a script long after it was originally written. It is recommended that you add comments throughout a script as you write it.

In SLAX, you insert comments in the traditional C style, beginning with **/*** and ending with ***/**. For example:

```
/* This is a comment. */
```

Multi-line comments follow the same format. In the following example, the additional "*" characters are added to the beginning of the lines for readability, but they are not required.

```
/* Script Title
* Author: Jane Doe
* Last modified: 01/01/10
* Summary of modifications: ...
*/
```

The XSLT equivalent is:

```
<!-- Script Title
 Author: Jane Doe
 Last modified: 01/01/10
 Summary of modifications: ...
-->
```

The following example inserts a comment into the script to remind the programmer that the output is sent to the console.

```
match / {
  <op-script-results> {
    /* Output script summary  to the console */
      <output> "Script summary: ...";
  }
}
```

Line Termination

As with many traditional programming languages, SLAX statements are terminated with a semicolon.

In the following example, the namespace declarations, import statement, and output element are all terminated with a semicolon. Lines that begin or end a block are not terminated with a semicolon.

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match / {
  <op-script-results> {
    <output> "Script summary:";
     /* ... */
  }
}
```

Strings

Strings are sequences of text characters. SLAX strings can be enclosed in either single quotes or double quotes. However, you must close the string with the same type of quote used to open the string. Strings can be concatenated together using the SLAX concatenation operation, which is the underscore (_).

For example:

```
match / {
  <op-script-results> {
    /* Output script summary  to the console */
      <output> "Script" _ "summary: ...";
  }
}
```

Related Documentation

- SLAX Elements and Element Attributes Overview on page 41
- SLAX Overview on page 35
- SLAX Statements Overview on page 51SLAX Templates Overview on page 43
- SLAX Templates Overview on page 43
- SLAX Variables Overview on page 50

SLAX Elements and Element Attributes Overview

SLAX Elements

SLAX elements are written with only the opening tag. The contents of the tag appear immediately following the opening tag. The contents can be either a simple expression or a more complex expression placed inside curly braces. For example:

```
<top> {
  <one>;
  <two> {
    <three>;
    <four>;
    <five> {
      <six>;
    }
  }
}
```

The XSLT equivalent is:

```
<top>
  <one/>
  <two>
    <three/>
    <four/>
    <five>
      <six/>
    </five>
  </two>
</top>
```

Using these nesting techniques and removing the closing tag reduces clutter and increases code clarity.

SLAX Element Attributes

SLAX element attributes follow the style of XML. Attributes are included in the opening tag and consist of an attribute name and value pair. The attribute syntax consists of the attribute name followed by an equals sign and then the attribute value enclosed in quotation marks. Multiple attributes are separated by spaces.

```
<element attr1="one" attr2="two">;
```

Where XSLT allows attribute value templates using curly braces, SLAX uses the normal expression syntax. Attribute values can include any XPath syntax, including quoted strings, parameters, variables, numbers, and the SLAX concatenation operator, which is an underscore (_). In the following example, the SLAX element **location** has two attributes, **state** and **zip**:

```
<location state=$location/state zip=$location/zip5 _ "-" _ $location/zip4>;
```

The XSLT equivalent is:

```
<location state="{$location/state}"
        zip="{concat($location/zip5, "-", $location/zip4}"/>
```

In SLAX, curly braces placed inside quote strings are not interpreted as attribute value templates. Instead, they are interpreted as plain-text curly braces.

An escape sequence causes a character to be treated as plain text and not as a special operator. For example, in HTML, an ampersand (**&**) followed by **lt** causes the less-than symbol (**<**) to be printed.

In XSLT, the double curly braces (**{{** and **}}**) are escape sequences that cause opening and closing curly braces to be treated as plain text. When a SLAX script is converted to XSLT, the curly braces inside quote strings are converted to double curly braces:

```
<avt sign="{here}">;
```

The XSLT equivalent is:

```
<avt sign="{{here}}"/>
```

Related Documentation

- XML Overview on page 11

XPath Expressions Overview for SLAX

XPath expressions can appear either as the contents of an XML element or as the contents of an **expr** (expression) statement. In either case, the value is translated to either an **<xsl:text>** element, which outputs literal text, or to an **<xsl:value-of>** element, which extracts data from an XML structure.

You encode strings using quotation marks (single or double). The concatenation operator is the underscore (_), as in PERL 6.

In this example, the contents of the **<three>** and **<four>** elements are identical, and the content of the **<five>** element differs only in the use of the XPath **concat()** function. The resulting output is the same in all three cases.

```
<top> {
```

```
    <one> "test";
    <two> "The answer is " _ results/answer _ ".";
    <three> results/count _ " attempts made by " _ results/user;
    <four> {
      expr results/count _ " attempts made by " _ results/user;
    }
    <five> {
      expr results/count;
      expr " attempts made by ";
      expr results/user;
    }
    <six> results/message;
  }
```

The XSLT equivalent is:

```
<top>
  <one><xsl:text>test</xsl:text></one>
  <two>
    <xsl:value-of select='concat("The answer is ", results/answer, ".")'/>
  </two>
  <three>
    <xsl:value-of select='concat(results/count, " attempts made by ", results/user)'/>
  </three>
  <four>
    <xsl:value-of select='concat(results/count, " attempts made by ", results/user)'/>
  </four>
  <five>
    <xsl:value-of select="results/count"/>
    <xsl:text> attempts made by </xsl:text>
    <xsl:value-of select="results/user"/>
  </five>
  <six><xsl:value-of select='results/message'/></six>
</top>
```

Related Documentation

- **concat() on page 95**
- SLAX Elements and Element Attributes Overview on page 41
- SLAX Syntax Rules Overview on page 39
- XPath Overview on page 21
- **xsl:text on page 106**
- **xsl:value-of on page 106**

SLAX Templates Overview

A SLAX script consists of one or more sets of rules called *templates*. Each template is a segment of code that contains rules to apply when a specified node is matched.

There are two types of templates, named and unnamed (or match), and they are described in the following sections.

Unnamed (Match) Templates

Unnamed templates, also known as match templates, contain a **match** statement with an XPath expression to specify the criteria for nodes upon which the template should be invoked. In the following commit script sample, the template matches the top-level element in the configuration hierarchy.

```
match configuration {
  /* ...body of the  template goes here */
}
```

By default, the processor recursively traverses the entire document hierarchy, inspecting each node, looking for a template that matches the current node. When a matching template is found, the contents of that template are evaluated.

The **apply-templates** statement can be used inside an unnamed template to limit and control the default, hierarchical traversal of nodes. This statement accepts an optional XPath expression, which is equivalent to the **select** attribute in an **<xsl:apply-templates>** element. If an optional XPath expression is included, only nodes matching the XPath expression are traversed. Otherwise all children of the context node are traversed. If the XPath expression is included but does not match any nodes, nothing is traversed and nothing happens.

In the following example, the template rule matches the **<route>** element in the XML hierarchy. All the nodes containing a **changed** attribute are processed. All **route** elements containing a **changed** attribute are replaced with a **new** element.

```
match route {
  <new> {
    apply-templates *[@changed];
  }
}
```

The XSLT equivalent:

```
<xsl:template match="route">
  <new>
    <xsl:apply-templates select="*[@changed]"/>
  </new>
</xsl:template>
```

Using unnamed templates allows the script to ignore the location of a tag in the XML hierarchy. For example, if you want to convert all **<author>** tags into ** ** tags, using templates enables you to write a single rule that converts all **<author>** tags, regardless of their location in the input XML document.

Named Templates

Named templates operate like functions in traditional programming languages. When the complexity of a script increases or a code segment appears in multiple places, you

can modularize the code and create named templates. Like functions, named templates accept arguments and run only when explicitly called.

In SLAX, the named template definition consists of the **template** keyword, the template name, a set of parameters, and a braces-delimited block of code. Parameter declarations can be inline and consist of the parameter name, and, optionally, a default value. Or you can declare parameters inside the template block using the **param** statement. If a default value is not defined, the parameter defaults to an empty string.

The following example creates a template named **my-template** and defines three parameters, one of which defaults to the string **false**, and one of which defaults to the contents of the element node named **name** that is a child of the current context node. If the script calls the template and does not pass in a parameter, the default value is used.

```
template my-template ($a, $b = "false", $c = name) {
  /* ... body of the template ... */
}
```

An alternate method is to declare the parameters within the template using the **param** statement. The following code is identical to the previous example:

```
template my-template {
  param $a;
  param $b = "false";
  param $c = name;
  /* ... body of the template ... */
}
```

In SLAX, you invoke named templates using the **call** statement, which consists of the **call** keyword and template name followed by a set of parameter bindings. These bindings are a comma-separated list of parameter names that are passed into the template from the calling environment. Parameter assignments are made by name and not by position in the list. Alternatively, you can declare parameters inside the **call** block using the **with** statement. Parameters passed into a template must match a parameter defined in the actual template; otherwise the parameter is ignored. Optionally, you can set a value for each parameter. If you do not define a value for the parameter in the calling environment, the script passes in the current value of the parameter if it was previously initialized, or it generates an error if the parameter was never declared. For more information about passing parameters, see “SLAX Parameters Overview” on page 47.

In the following example, the template **my-template** is called with the parameter **c** containing the contents of the element node named **other-name** that is a child of the current context node.

```
call my-template {
  with $c = other-name;
}
```


In the following example, the **name-servers-template** declares two parameters **name-servers** and **size**. The **size** parameter is given a default value of zero. The match template, which declares and initializes **$name-servers**, calls the **name-servers-template** three times.

The first call to the template does not include any parameters. Thus **name-servers** will default to an empty string, and **size** will default to a value of zero as defined in the template. The second call includes the **name-servers** and **size** parameters but only supplies a value for the **size** parameter. Thus **name-servers** has the value defined by its initialization in the script, and **size** is equal to the number of **name-servers** elements in the configuration hierarchy. The last call is identical to the second call, but it supplies the parameters using the **with** statement syntax.

```
match configuration {
  param $name-servers = name-servers/name;
  call name-servers-template();
  call name-servers-template($name-servers, $size = count($name-servers));
  call name-servers-template() {
    with $name-servers;
    with $size = count($name-servers);
  }
}
  template name-servers-template($name-servers, $size = 0) {
    <output> "template called with size " _ $size;
  }
```

The XSLT equivalent is:

```
<xsl:template match="configuration">
  <xsl:variable name="name-servers" select="name-servers/name"/>
  <xsl:call-template name="name-servers-template"/>
  <xsl:call-template name="name-servers-template">
    <xsl:with-param name="name-servers" select="$name-servers"/>
    <xsl:with-param name="size" select="count($name-servers)"/>
  </xsl:call-template>
  <xsl:call-template name="name-servers-template">
    <xsl:with-param name="name-servers" select="$name-servers"/>
    <xsl:with-param name="size" select="count($name-servers)"/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="name-servers-template">
  <xsl:param name="name-servers"/>
  <xsl:param name="size" select="0"/>
  <output>
    <xsl:value-of select="concat('template called with size ', $size)"/>
  </output>
</xsl:template>
```

Related Documentation

- SLAX Parameters Overview on page 47
- XSLT Templates Overview on page 24
- **apply-templates on page 110**
- **call on page 111**
- **match on page 115**

- param on page 117
- with on page 122

SLAX Parameters Overview

Parameters can be passed to either named or unnamed templates. Inside the template, parameters must be declared and can then be referenced by prefixing their name with the dollar sign ($).

Declaring Parameters

In SLAX, you declare parameters using the **param** statement. Optionally, you can define an initial value for each parameter in the declaration. For example:

```
param $dot = .;
```

The scope of a parameter can be global or local. A parameter whose value is set by Junos at script initialization must be defined as a global parameter. Global parameter declarations are placed just after the stylesheet declarations. A script can assign a default value to the global parameter, which is used in the event that Junos does not give a value to the parameter.

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns ext = "http://xmlsoft.org/XSLT/namespace";

/* global parameter */
param $interface1 = “fxp0”;
```

Local parameters must be declared at the beginning of a block and their scope is limited to the block in which they are declared. In a template, you declare parameters either in a parameter list or by using the **param** statement in the template block. Optionally, declare default values for each template parameter. If a template is invoked without the parameter, the default expression is evaluated, and the results are assigned to the parameter. If you do not define a default value in the template, the parameter defaults to an empty string.

The following named template **print-host-name** declares the parameter **message** and defines a default value:

```
template print-host-name ($message = "host name: " _ system/host-name) {
  <xnm:warning> {
    <message> $message;
  }
}
```

An alternative, but equivalent, declaration is:

```
template print-host-name () {
  param $message = "host name: " _ system/host-name;
  <xnm:warning> {
    <message> $message;
```

```
  }
}
```

The template declares **message** and accesses its value by prefixing the parameter name with the dollar sign ($). In XSLT, the parameter name is prefixed by the dollar sign when you access it but not when you declare it.

Passing Parameters

When you invoke a template, you pass arguments into the template either in an argument list or by using the **with** statement. The name of the parameter supplied in the calling environment must match the name of a parameter defined in the actual template; otherwise the parameter is ignored. Optionally, for each parameter you pass to a template, you can define a value using an equal sign (=) and a value expression. In the following example, the two calls to the named template **print-host-name** are identical:

```
match configuration {
   call print-host-name($message = "passing in host name: " _ system/host-name);
}
match configuration {
   call print-host-name() {
      with $message = "passing in host name: " _ system/host-name;
   }
}
```

The parameter value that gets used in a template depends on how the template is called. The following three examples, which call the **print-host-name** template, illustrate the possible calling environments.

If you call a template but do not include a specific parameter, the default expression defined in the template is evaluated, and the results are assigned to the parameter. If there is no default value for that parameter in the template, the parameter defaults to an empty string. The following example calls the named template **print-host-name** but does not include any parameters in the call. In this case, the named template will use the default value for the **message** parameter that was defined in the **print-host-name** template, or an empty string if no default exists.

```
match configuration {
   call print-host-name();
}
```

If you call a template and include a parameter, but do not define a value for the parameter in the calling environment, the script passes in the current value of the parameter if it was previously initialized, or it generates an error if the parameter was never declared. The following example calls the named template **print-host-name** and passes in the **message** parameter but does not include a value. If **message** is declared and initialized in the script, and the scope is visible to the block, the current value of **message** is used. If **message** is declared in the script but not initialized, the value of **message** will be an empty string. If **message** has not been declared, the script produces an error.

```
match configuration {
   call print-host-name($message);
   /* If $message was initialized previously, the current value is used;
```

```
    * If $message was declared but not initialized, an empty string is used;
    * If $message was never declared, the call generates an error. */
}
```

If you call a template, include the parameter, and define a value for the parameter, the template uses the provided value. The following example calls the named template **print-host-name** with the **message** parameter and a defined value, so the template uses the new value.

```
match configuration {
  call print-host-name($message = "passing in host name: " _ system/host-name);
}
```

Example: Parameters and Match Templates

The following example matches the top level **configuration** hierarchy element and then instructs the Junos OS management process (mgd) to recursively apply templates to the **system/host-name** subtree. The parameters **message** and **domain** are used in the processing of any matching nodes.

```
match configuration {
  var $domain = domain-name;
  apply-templates system/host-name {
    with $message = "Invalid host-name";
    with $domain;
  }
}

match host-name {
  param $message = "Error";
  param $domain;
  <hello> $message _ ":: " _ . _ " (" _ $domain _ ")";
}
```

The XSLT equivalent is:

```
<xsl:template match="configuration">
  <xsl:apply-templates select="system/host-name">
    <xsl:with-param name="message" select="'Invalid host-name'"/>
    <xsl:with-param name="domain" select="$domain"/>
  </xsl:apply-templates>
</xsl:template>

<xsl:template match="host-name">
  <xsl:param name="message" select="'Error'"/>
  <xsl:param name="domain"/>
  <hello>
    <xsl:value-of select="concat($message, ':: ', ., ' (', $domain, ')')"/>
  </hello>
</xsl:template>
```

Related Documentation

- SLAX Templates Overview on page 43
- **param on page 117**
- **template on page 119**

- with on page 122

SLAX Variables Overview

In SLAX, you declare variables using the **var** statement. In the declaration, the variable name is prefixed with the dollar sign ($), unlike the XSLT declaration, where the dollar sign does not prefix the value of the **name** attribute of the **<xsl:variable>** element. Once you declare a variable, you can reference it within an XPath expression using the variable name prefixed with a dollar sign ($).

Variables are immutable; you can set the value of a variable only when you declare the variable, after which point, the value is fixed. You initialize a variable by following the variable name with an equal sign (=) and an expression. The following example declares and initializes the variable **location**. The **location** variable is then used to initialize the **message** variable.

```
var $location = $dot/@location;
var $message = "We are in " _ $location _ " now.";
```

The XSLT equivalent is:

```
<xsl:variable name="location" select="$dot/@location"/>
<xsl:variable name="message" select="concat('We are in ', $location, ' now.')"/>
```

You can define both local and global variables. Variables are global if they are defined outside of any template. Otherwise, they are local. The value of a global variable is accessible anywhere in the stylesheet. The scope of a local variable is limited to the template or code block in which it is defined.

SLAX variables can store any values that you can calculate or statically define. This includes data structures, XML hierarchies, and combinations of text and parameters. For example, you could assign the XML output of an operational mode command to a variable and then access the hierarchy within the variable.

Variables are immutable. As such, you can never change the value of a variable after it is defined in the declaration. Although you cannot directly update the value of a variable, you can mimic the effect by recursively calling a function and passing in the value of the variable as a parameter. For example:

```
var $count = 1;
match / {
  call update-count($myparam = $count);
}
template update-count($myparam) {
  expr $count _ ", " $myparam _"\n";
  if ($myparam != 4) {
    call update-count($myparam = $myparam + 1)
  }
}
```

Executing the op script in the CLI produces the following output in the log file. Although the **count** variable must remain fixed, **myparam** is updated with each call to the template.

```
1, 1
1, 2
```



```
1, 3
1, 4
1, 5
```

Related Documentation

- XSLT Variables Overview on page 29
- SLAX Parameters Overview on page 47
- **var on page 120**

SLAX Statements Overview

This section lists some commonly used SLAX statements, with brief examples and XSLT equivalents.

- for-each Statement on page 51
- if, else if, and else Statements on page 52
- match Statement on page 53
- ns Statement on page 53
- version Statement on page 54

for-each Statement

The SLAX **for-each** statement functions like the **<xsl:for-each>** element. The statement consists of the **for-each** keyword, a parentheses-delimited expression, and a curly braces-delimited block. The **for-each** statement tells the processor to gather together a set of nodes and process them one by one. The nodes are selected by the specified XPath expression. Each of the nodes is then processed according to the instructions held in the **for-each** code block.

```
for-each (xpath-expression) {
  ...
}
```

Code inside the **for-each** instruction is evaluated recursively for each node that matches the XPath expression. That is, the current context is moved to each node selected by the **for-each** clause, and processing is relative to that current context.

In the following example, the **inventory** variable stores the inventory hierarchy. The **for-each** statement recursively processes each **chassis-sub-module** node that is a child of **chassis-module** that is a child of the **chassis** node. For each **chassis-sub-module** element that contains a **part-number** with a value equal to the specified part number, a **message** element is created that includes the name of the chassis module and the name and description of the chassis sub module.

```
for-each ($inventory/chassis/chassis-module/
        chassis-sub-module[part-number == '750-000610']) {
  <message> "Down rev PIC in " _ ../name _ ", " _ name _ ": " _ description;
}
```

The XSLT equivalent is:

```
<xsl:for-each select="$inventory/chassis/chassis-module/
         chassis-sub-module[part-number = '750-000610']">
```

```
    <message>
      <xsl:value-of select="concat('Down rev PIC in ', ../name, ', ', name, ': ',
          description)"/>
    </message>
  </xsl:for-each>
```

if, else if, and else Statements

SLAX supports **if**, **else if**, and **else** statements. The **if** statement is a conditional construct that causes instructions to be processed if the specified XPath expression evaluates to true. The **if** construct may have one or more associated **else if** clauses, each of which tests an XPath expression. If the expression in the **if** statement evaluates to false, the processor checks each **else if** expression. If a statement evaluates to true, the script executes the instructions in the associated block and ignores all subsequent **else if** and **else** statements. The optional **else** clause is the default code that is executed in the event that all associated **if** and **else-if** expressions evaluate to false. If all of the **if** and **else if** statement evaluate to false, and the **else** statement is not present, no action is taken.

The expressions that appear in parentheses are extended XPath expressions, which support the double equal sign (**==**) in place of XPath's single equal sign (**=**).

```
if (expression) {
  /* If block Statement */
}
else if (expression) {
   /* else if block statement */
}
else {
  /* else block statement */
}
```

During script processing, an **if** statement that does not have an associated **else if** or **else** statement is transformed into an **<xsl:if>** element. If either the **else if** or **else** clauses are present, the **if** statement and associated **else if** and **else** blocks are transformed into an **<xsl:choose>** element.

```
if (starts-with(name, "fe-")) {
  if (mtu < 1500) {
    /* Select Fast Ethernet interfaces with low MTUs */
  }
}
else {
  if (mtu > 8096) {
    /* Select non-Fast Ethernet interfaces with high MTUs */
  }
}
```

The XSLT equivalent is:

```
<xsl:choose>
  <xsl:when test="starts-with(name, 'fe-')">
    <xsl:if test="mtu &lt; 1500">
      <!-- Select Fast Ethernet interfaces with low MTUs -->
    </xsl:if>
  </xsl:when>
  <xsl:otherwise>
    <xsl:if test="mtu &gt; 8096">
      <!-- Select non-Fast Ethernet interfaces with high MTUs -->
```



```
    </xsl:if>
  </xsl:otherwise>
</xsl:choose>
```

match Statement

You specify basic match templates using the **match** statement, followed by an expression specifying when the template should be allowed and a block of statements enclosed in a set of braces.

```
match configuration {
   <xnm:error> {
     <message> "...";
  }
}
```

The XSLT equivalent is:

```
<xsl:template match="configuration">
  <xnm:error>
    <message> ...</message>
  </xnm:error>
</xsl:template>
```

For more information about constructing match templates, see “SLAX Templates Overview” on page 43.

ns Statement

You specify namespace definitions using the SLAX **ns** statement. This consists of the **ns** keyword, a prefix string, an equal sign, and a namespace Uniform Resource Identifier (URI). To define the default namespace, use only the **ns** keyword and a namespace URI.

```
ns junos = "http://www.juniper.net/junos/";
```

The **ns** statement can appear after the **version** statement at the beginning of the style sheet or at the beginning of any block.

```
ns a = "http://example.com/1";
ns "http://example.com/global";
ns b = "http://example.com/2";
match / {
  ns c = "http://example.com/3";
  <top> {
    ns a = "http://example.com/4";
    apply-templates commit-script-input/configuration;
  }
}
```

When it appears at the beginning of the style sheet, the **ns** statement can include either the **exclude** or **extension** keyword. The keyword instructs the parser to add the namespace prefix to the **exclude-result-prefixes** or **extension-element-prefixes** attribute.

```
ns exclude foo = "http://example.com/foo";
ns extension jcs = "http://xml.juniper.net/jcs";
```

The XSLT equivalent is:

```
<xsl:stylesheet xmlns:foo="http://example.com/foo"
```

```
  xmlns:jcs="http://xml.juniper.net/jcs"
  exclude-result-prefixes="foo"
  extension-element-prefixes="jcs">
  <!-- ... -->
</xsl:stylesheet>
```

version Statement

All SLAX style sheets must begin with a **version** statement, which specifies the version number for the SLAX language. The current version is **1.0**. SLAX version 1.0 uses XML version 1.0 and XSLT version 1.1.

```
version 1.0;
```

The XSLT equivalent is:

```
<xsl:stylesheet version="1.0">
```

Related Documentation

- **else on page 112**
- **for-each on page 113**
- **if on page 114**
- **match on page 115**
- **version on page 121**

XSLT Elements Without SLAX Equivalents

Some XSLT elements are not directly translated into SLAX statements. Some examples of XSLT elements for which there are no SLAX equivalents are **<xsl:fallback>**, **<xsl:output>**, and **<xsl:sort>**.

You can encode these elements directly as normal SLAX elements in the XSLT namespace. For example, you can include the **<xsl:output>** and **<xsl:sort>** elements in a SLAX script, as shown here:

```
<xsl:output method="xml" indent="yes" media-type="image/svg">;
match * {
  for-each (configuration/interfaces/unit) {
    <xsl:sort order="ascending">;
  }
}
```

When you include XSLT namespace elements in a SLAX script, do not include closing tags. For empty tags, do not include a forward slash (/) after the tag name. The examples shown in this section demonstrate the correct syntax.

The following XSLT snippet contains a combination of elements, some of which have SLAX counterparts and some of which do not:

```
<xsl:loop select="title">
  <xsl:fallback>
    <xsl:for-each select="title">
      <xsl:value-of select="."/>
    </xsl:for-each>
```

```
    </xsl:fallback>
  </xsl:loop>
```

The SLAX conversion uses the XSLT namespace for XSLT elements that do not have SLAX counterparts:

```
<xsl:loop select = "title"> {
  <xsl:fallback> {
    for-each (title) {
      expr .;
    }
  }
}
```

SLAX Operators

SLAX provides a variety of operators, which add great versatility to the SLAX scripting language. Table 6 on page 55 summarizes the available operators and provides an example and an explanation of each.

Table 6: SLAX Operators

Name	Operator	Example / Explanation
Addition	+	var $example = 1 + 1; Assigns the value of 1 + 1 to the **$example** variable.
Subtraction, Negation	-	var $example = 1 - 1; Assigns the value of 1 - 1 to the **$example** variable; when used as an unary operator, it changes the sign of a number from positive to negative or from negative to positive.
Multiplication	*	<output>5 * 10; Results in the value 50 being written to the console.
Division	div	<output>$bit-count div 8; Divides the bits by eight, returning the byte count, and displays the result on the console (requires that **$bit-count** has been initialized).
Modulo	mod	<output>10 mod 3; Returns the division remainder of two numbers. In this example, the expression writes 1 to the console.
Equals	==	$mtu == 1500 If $mtu is 1500 then the expression resolves to true, otherwise it returns false (requires that **$mtu** has been initialized).
Does not equal	!=	$mtu != 1500 If **$mtu** equals 1500 then the result is false, otherwise it returns true (requires that **$mtu** has been initialized)

Table 6: SLAX Operators *(continued)*

Name	Operator	Example / Explanation
Less than	<	$hop-count < 15 Returns true if the value to the left of the operator is less than the value to the right, otherwise it returns false. In this example, if $hop-count is less than 15, the expression returns true (requires that **$hop-count** has been initialized).
Less than or equal to	<=	$hop-count <= 14 Returns true if the value to the left of the operator is either less than the value to the right or equal to it, otherwise it returns false. In this example, if **$hop-count** is 14 or less, the expression returns true (requires that **$hop-count** has been initialized).
Greater than	>	$hop-count > 0 Returns true if the value to the left of the operator is greater than the value to the right, otherwise it returns false. In this example, if **$hop-count** is greater than zero, the expression returns true (requires that **$hop-count** has been initialized).
Greater than or equal to	>=	$hop-count >= 1 Returns true if the value to the left of the operator is either greater than the value to the right or equal to it, otherwise it returns false. In this example, if **$hop-count** is one or greater, the expression returns true (requires that **$hop-count** has been initialized).
Parentheses	( )	var $result = ($byte-count * 8) + 150; Used to create complex expressions. Parentheses function the same way as in a mathematical expression, with the expression within the parentheses evaluated first. Parentheses can be nested; the innermost set of parentheses is evaluated first, then the next set, and so on.
And	&&	$byte-count > 500000 && $byte-count < 1000000 Evaluates two expressions and returns one boolean result. If either of the two expressions evaluates to false, then the combined expression evaluates to false.
Or	||	$mtu-size != 1500 || $mtu-size > 2000 Evaluates two expressions and returns one boolean result. If either of the two expressions evaluates to true, then the combined expression evaluates to true.
String concatenation	_ (under-score)	var $combined-string = $host-name _ " is located at " _ $location; Concatenates multiple strings (note that strings cannot be combined using the + operator in SLAX). In the example, if **$host-name** is "r1" and **$location** is "HQ" then the value of **$combined-string** is "r1 is located at HQ".

Table 6: SLAX Operators *(continued)*

Name	Operator	Example / Explanation
Node-Set Union	|	var $all-interface-nodes = $fe-interface-nodes | $ge-interface-nodes; Creates a union of two node-sets. All the nodes from one set combine with the nodes in the second set. This is useful when a script needs to perform a similar operation over XML nodes that are pulled from multiple sources.
Result Tree Fragment to Node-Set Conversion	:=	var $new-node-set := $rtf-variable; A result tree fragment contains an unparsed XML data structure. It is not possible to retrieve any of the embedded XML information from this data type. The **:=** operator converts a variable from a result tree fragment into a node-set. The script can then tell the Junos OS to search the node-set for the appropriate information and extract it. Only Junos OS Release 9.2 and beyond support this operator.

Related Documentation

- SLAX Elements and Element Attributes Overview on page 41
- SLAX Overview on page 35
- SLAX Statements Overview on page 51
- SLAX Syntax Rules Overview on page 39
- SLAX Variables Overview on page 50

CHAPTER 6

Junos Script Automation: Extension Functions, Templates, and Parameters

Junos OS automation features include a library of extension functions and predefined templates and parameters that can be used in all scripts. The library includes extension functions that accomplish scripting tasks more easily, and an import file, junos.xsl, which includes named templates and predefined parameters that make scripts easier to read and write. This chapter discusses the extension functions, templates, and parameters in detail in the following sections:

Junos Script Automation: Extension Functions in the jcs Namespace Overview

The Junos OS extension functions are used in commit, op, and event scripts to accomplish scripting tasks more easily. Extension functions allow you to perform operations that are difficult or impossible to perform in XPath. The library provides logic, data manipulation, input and output, and utility functions.

The Junos OS extension functions are defined in the namespace with the associated Uniform Resource Identifier (URI) http://xml.juniper.net/junos/commit-scripts/1.0 . This avoids name conflicts with standard XSLT functions. To use the Junos extension functions in scripts, you must include the namespace URI in your style sheet declaration. Generally, the **jcs** prefix is mapped to the URI, and you then use the extension functions by prepending the **jcs** prefix to the function name. During processing, the **jcs:** prefix is expanded into the URI reference.

To call an extension function in a script, you include any required variable declarations, a variable call with the **select="jcs:*function-name*()"** attribute for XSLT scripts or a simple function call for SLAX scripts, and pass along any required or optional arguments. Arguments must be passed into the function in the precise order specified by the function definition. This is different from a template, where the parameters are assigned by name and can appear in any order. The return value of an extension function must always either be assigned to a variable or designated as output.

In the following example, the **jcs** prefix is mapped to the namespace identified by the URI http://xml.juniper.net/junos/commit-scripts/1.0, which defines the extension functions used in commit, op, and event scripts.The script then calls the **jcs:invoke()** function with one argument.

XSLT Syntax

```
<?xml version=”1.0”?>
<xsl:stylesheet version=”1.0”
    xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
    ...
    <xsl:variable name="result" select="jcs:invoke($command)"/>
    ...
</xsl: stylesheet>
```

SLAX Syntax

```
version 1.0;
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";...
    ...
    var $result = jcs:invoke($command);
    ...
```

Related Documentation

- Junos Script Automation: Named Templates in the jcs Namespace Overview on page 80
- Junos Script Automation: Predefined Parameters in the junos.xsl File on page 92
- Junos Extension Functions in the jcs Namespace Summary on page 60
- SLAX Variables Overview on page 50
- XSLT Variables Overview on page 29

Junos Extension Functions in the jcs Namespace Summary

The Junos extension functions are summarized in the following table:

Table 7: Junos Extension Functions

Function	Type	Description
jcs:break-lines()	Data manipulation	Break a simple element into multiple elements, delimited by newlines.
jcs:close()	Utility	Close a previously opened connection handle.
jcs:dampen()	Utility	Prevent the same operation from being repeatedly executed within a script.

Table 7: Junos Extension Functions *(continued)*

Function	Type	Description
jcs:empty()	Logic	Evaluate a node set or string argument to determine if it is an empty value.
jcs:execute()	Utility	Execute a remote procedure call (RPC) within the context of a specified connection handle.
jcs:first-of()	Logic	Return the first nonempty (non-null) item in a list, If all objects in the list are empty, the default expression is returned.
jcs:get-input()	Input/output control	Invoke a CLI prompt and wait for user input. If the script is run non-interactively, the function returns an empty value. This function cannot be used with event scripts.
jcs:get-secret()	Input/output control	Invoke a CLI prompt and wait for user input. The input is not echoed back to the user.
jcs:hostname()	Utility	Return the fully qualified domain name associated with a given IPv4 or IPv6 address, provided the DNS server is configured on the router.
jcs:invoke()	Utility	Invoke a remote procedure call (RPC) on the local device.
jcs:open()	Utility	Return a connection handle that can be used to execute remote procedure calls (RPCs).
jcs:output()	Input/output control	Generate unformatted output text that is immediately sent to the CLI session.
jcs:parse-ip()	Data manipulation	Parse an IPv4 or IPv6 address and return the host IP address, protocol family, prefix length, network address, and network mask.
jcs:printf()	Input/output control	Generate formatted output text. Most standard **printf** formats are supported, in addition to some Junos OS–specific formats. The function returns a formatted string but does not print it on call.
jcs:progress()	Input/output control	Issue a progress message containing the single argument immediately to the CLI session provided that the **detail** flag was specified when the script was invoked.
jcs:regex()	Data manipulation	Evaluate a regular expression against a given string argument and return any matches.
jcs:sleep()	Utility	Cause the script to sleep for a specified time.
jcs:split()	Data manipulation	Split a string into an array of substrings delimited by a regular expression pattern.

Table 7: Junos Extension Functions *(continued)*

Function	Type	Description
jcs:sysctl()	Utility	Return the value of the given **sysctl** value as a string or an integer.
jcs:syslog()	Input/output control	Log messages with the specified priority to the system log file.
jcs:trace()	Input/output control	Issue a trace message, which is sent to the trace file.

Related Documentation

- Junos Named Templates in the jcs Namespace Summary on page 81
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- Junos Script Automation: Named Templates in the jcs Namespace Overview on page 80
- Junos Script Automation: Predefined Parameters in the junos.xsl File on page 92

Junos Extension Functions in the jcs Namespace

The Junos extension functions are discussed in detail in the following sections:

- **jcs:syslog( ) Function on page 78**
- **jcs:trace( ) Function on page 80**

jcs:break-lines( ) Function

Syntax

```
var $lines = jcs:break-lines(expression);
<xsl:variable name="lines" select="jcs:break_lines(expression)"/>
```

Description Break a simple element into multiple elements, delimited by newlines. This is especially useful for large output elements such as those returned by the **show pfe** command.

Parameters

- ***expression***—Original output.

Return Value

- **$lines**—Output broken up into lines.

Usage Examples

```
var $lines = jcs:break-lines($output);
for-each ($lines) {
  ...
}
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:parse-ip( ) Function on page 72**
- **jcs:regex( ) Function on page 75**
- **jcs:split( ) Function on page 76**

jcs:close( ) Function

Syntax

```
var $results =  jcs:close(connection);
<xsl:variable name="results" select="jcs:close(connection)"/>
```

Description Close a previously opened connection handle.

Parameters

- ***connection***—a connection handle generated by a call to the **jcs:open( )** function.

Usage Examples The following example closes the connection handle **$connection**, which was originally generated by a call to the **jcs:open( )** function:

```
var $connection = jcs:open( );
var $result = jcs:close($connection);
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59

- jcs:execute( ) Function on page 65
- jcs:open( ) Function on page 70

jcs:dampen( ) Function

Syntax

var $result = jcs:dampen(*tag-string*, *max*, *interval*);

<xsl:variable name="result" select="jcs:dampen(*tag-string*, *max*, *interval*)"/>

Description

Prevent the same operation from being repeatedly executed within a script. The dampen function returns **false** if the number of calls to the **jcs:dampen( )** function exceeds a ***max*** number of calls in the time interval ***interval***. Otherwise the function returns **true**. The function parameters include an arbitrary string that is used to distinguish different calls to the **jcs:dampen( )** function. This tag is stored in the **/var/run** directory on the device.

Parameters

- ***interval***—Time interval, in minutes.
- ***max***—Maximum number of calls to the **jcs:dampen( )** function with a given tag allowed before the function returns **false**. This limit is based on the number of calls within a specified time interval.
- ***tag-string***—Arbitrary string used to distinguish different calls to the **jcs:dampen( )** function.

Return Value

- **result**—Boolean value based on the number of calls to **jcs:dampen( )** with a given tag and within a specified time. If the number of calls for a given tag exceeds ***max***, the return value is **false**. If the number of calls is less than ***max***, the return value is **true**.

Usage Examples

In the following example, if the **jcs:dampen( )** function with the tag **'mytag1'** is called less than three times in a 10-minute interval, the function returns **true**. If the function is called more than three times within 10 minutes, the function returns **false**.

```
if (jcs:dampen('mytag1', 3, 10)) {
   /* Code for situations when jcs:dampen( ) with */
   /* the tag 'mytag1' is called less than three times */
  /* within 10 minutes */
} else {
  /* Code for situations when jcs:dampen( ) with */
   /* the tag 'mytag1' exceeds the three call maximum */
  /* limit within 10 minutes */
}
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59

jcs:empty( ) Function

Syntax

var $result = jcs:empty(*node-set* | *string*);

<xsl:variable name="result" select="jcs:empty(*node-set* | *string*)"/>

Description Test for the presence of a value and return **true** if the node set or string argument evaluates to an empty value.

Parameters
- *(node-set | string)*—Argument to test for the presence of a value.

Return Value
- **result**—Boolean value that is **true** if the argument is empty.

Usage Examples The following example shows that if **$set** is empty, the script will execute the enclosed code block.

```
if ( jcs:empty($set) ) {
  /* Code to handle true value ($set is empty) */
}
```

The following example tests whether the **description** node for interface fe-/0/0/0 is empty. If the description is missing, a message tag is output.

```
if (jcs:empty(interfaces/interface[name="fe-0/0/0"]/description)) {
  <message> "interface" _ name _ "is missing description";
}
```

Related Documentation
- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:first-of( ) Function on page 66**

jcs:execute( ) Function

Syntax

```
var $result = jcs:execute(connection, rpc);
```

```
<xsl:variable name="result" select="jcs:execute(connection, rpc)"/>
```

Description Execute a remote procedure call (RPC) within the context of a specified connection handle. Any number of RPCs may be executed within the context of the connection handle until it is closed with the **jcs:close( )** function.

Parameters
- ***connection***—Connection handle generated by a call to the **jcs:open( )** function.
- ***rpc***—Remote procedure call (RPC) to be executed.

Return Value

- **result**—Results of the executed RPC, which includes the contents of the **<rpc-reply>** element, but not the **<rpc-reply>** tag itself. This **$result** variable is the same as that produced by the **jcs:invoke()** function.

Usage Examples

In the following example, the **$rpc** variable is declared and initialized with the Junos XML **<get-interface-information>** element. A call to the **jcs:open()** function generates a connection handle to the local device. The code calls **jcs:execute()** and passes in the connection handle and RPC as arguments. The Junos OS processes the RPC and returns the results, which are stored in the **$results** variable.

```
var $rpc = <get-interface-information>;
var $connection = jcs:open();
var $results = jcs:execute($connection, $rpc);
expr $results;
```

In XSLT:

```
<xsl:variable name="connection" select="jcs:open()"/>
<xsl:variable name="rpc">
  <get-interface-information/>
</xsl:variable>
<xsl:variable name="results" select="jcs:execute($connection, $rpc)"/>
<xsl:value-of select="$results"/>
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:close() Function on page 63**
- **jcs:invoke() Function on page 69**
- **jcs:open() Function on page 70**

jcs:first-of() Function

Syntax

```
var $result = jcs:first-of(object, "expression");
```

```
<xsl:variable name="result" select="jcs:first-of(object, 'expression')"/>
```

Description

Return the first nonempty (non-null) item in a list. This function provides the same functionality as an **if / else-if / else** construct but in a much more concise format. If all objects in the list are empty, the default expression is returned.

Parameters

- ***expression***—Default value returned if all objects in the list are empty.
- ***object***—List of objects.

Return Value

- **result**—First nonempty (non-null) item in the object list. If all objects in the list are empty, the default expression is returned.

Usage Examples

In the following example, if the value of **a** is empty, **b** is checked. If the value of **b** is empty, **c** is checked. If the value of **c** is empty, **d** is checked. If the value of **d** is empty, the string "**none**" is returned.

```
jcs:first-of($a, $b, $c, $d, "none")
```

In the following example, for each physical interface, the script checks for a description of each logical-interface. If a logical interface description does not exist, the function returns the description of the (parent) physical interface. If the parent physical interface description does not exist, the function returns a message that no description was found.

```
var $rpc = <get-interface-information>;
var $results = jcs:invoke($rpc);
for-each ($results/physical-interface/logical-interface) {
  var $description = jcs:first-of(description, ../description, "no description found");
}
```

The equivalent XSLT code is:

```
<xsl:variable name="rpc">
  <get-interface-information/>
</xsl:variable>
<xsl:variable name="results" select="jcs:invoke($rpc)"/>
<xsl:for-each select="$results/physical-interface/logical-interface">
  <xsl:variable name="description"
     select="jcs:first-of(description, ../description, 'no description found')"/>
</xsl:for-each>
```

The code for the **description** variable in the previous examples would be equivalent to the following more verbose **if / else-if / else** construct:

```
var $description = {
  if (description) {
    expr description;
  }
  else if (../description) {
    expr ../description;
  }
  else {
    expr "no description found";
  }
}
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:empty( ) Function on page 64**

jcs:get-input( ) Function

Syntax

```
var $user-input =  jcs:get-input(string);
```

```
<xsl:variable name="user-input" select="jcs:get-input(string)"/>
```

Description Invoke a CLI prompt and wait for user input. The user input is defined as a string for subsequent use. If the script is run non-interactively, the function returns an empty value. This function cannot be used with event scripts.

Parameters
- ***string***—CLI prompt text.

Return Value
- **user-input**—Text typed by the user and stored as a string. The return value will be empty if the script is run non-interactively.

Usage Examples In the following example, the user is prompted to enter a login name. The user's input is stored in the variable **$username**:

```
var $username = jcs:get-input("Enter login id: ");
```

Related Documentation
- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:get-secret() Function on page 68**
- **jcs:output() Function on page 71**
- **jcs:printf() Function on page 73**
- **jcs:progress() Function on page 74**
- **jcs:syslog() Function on page 78**
- **jcs:trace() Function on page 80**

jcs:get-secret() Function

Syntax

```
var $user-input = jcs:get-secret(string);
<xsl:variable name="user-input" select="jcs:get-secret(string)"/>
```

Description Invoke a CLI prompt and wait for user input. Unlike the **jcs:get-input()** function, the input is not echoed back to the user, which makes the function useful for obtaining passwords. The user input is defined as a string for subsequent use. This function cannot be used with event scripts.

Parameters
- ***string***—CLI prompt text.

Return Value
- **user-input**—Text typed by the user and stored as a string.

Usage Examples The following example shows how to prompt for a password that is not echoed back to the user:

```
var $password = jcs:get-secret("Enter password: ");
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:get-input() Function on page 67**
- **jcs:output() Function on page 71**
- **jcs:printf() Function on page 73**
- **jcs:progress() Function on page 74**
- **jcs:syslog() Function on page 78**
- **jcs:trace() Function on page 80**

jcs:hostname() Function

Syntax

```
var $name = jcs:hostname(expression);
<xsl:variable name="name" select="jcs:hostname(expression)"/>
```

Description Return the fully qualified domain name associated with a given IPv4 or IPv6 address. The DNS server must be configured on the router in order to resolve the domain name.

Parameters

- ***expression***—IPv4 or IPv6 address.

Return Value

- **name**—Hostname associated with the IP address.

Usage Examples

```
<xsl:variable name="host-name" select="jcs:hostname($dest-address)"/>
<xsl:value-of select="concat($address,' is ', jcs:hostname($address))"/>
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:invoke() Function on page 69**
- **jcs:parse-ip() Function on page 72**
- **jcs:sleep() Function on page 76**

jcs:invoke() Function

Syntax

```
var $result = jcs:invoke(rpc);
<xsl:variable name="result" select="jcs:invoke(rpc)"/>
```

Description Invoke a remote procedure call (RPC) on the local device. The function can be called with one argument, either a string containing a Junos XML or Junos XML protocol RPC method name or a tree containing an RPC. The result is the contents of the **<rpc-reply>** element, not including the **<rpc-reply>** tag. The RPCs allow you to perform functions equivalent to the Junos OS operational mode commands.

Parameters

- *rpc*—String containing a Junos XML or Junos XML protocol RPC method name or a tree containing an RPC.

Return Value

- **result**—Results of the executed RPC, which includes the contents of the **<rpc-reply>** element, but not the **<rpc-reply>** tag itself.

Usage Examples

In the following example, there is a test to see if the **interface** argument is included on the command line when the script is executed. If the argument is provided, the operational mode output of the **show interfaces terse** command is narrowed to include information about that interface only.

```
<xsl:param name="interface"/>
<xsl:variable name="rpc">
  <get-interface-information>
    <terse/>
    <xsl:if test="$interface">
      <interface-name>
        <xsl:value-of select="$interface"/>
      </interface-name>
    </xsl:if>
  </get-interface-information>
</xsl:variable>
<xsl:variable name="out" select="jcs:invoke($rpc)"/>
```

In this example, the **jcs:invoke()** function calls an RPC without modifying the output:

```
<xsl:variable name="sw" select="jcs:invoke('get-software-information')"/>
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:execute() Function on page 65**
- **jcs:hostname() Function on page 69**
- **jcs:sleep() Function on page 76**

jcs:open() Function

Syntax

```
var $connection = jcs:open(remote-hostname, username, passphrase);
```

```
<xsl:variable name="connection" select="jcs:open(remote-hostname, username,
passphrase)"/>
```

Description

Return a connection handle that can be used to execute remote procedure calls (RPCs) using the **jcs:execute()** extension function. To execute an RPC on a remote device, an SSH session must be established. In order for the script to establish the connection, you must either configure the SSH host key information for the remote device on the local device where the script will be executed, or the SSH host key information for the remote device must exist in the known hosts file of the user executing the script. The connection handle is closed with the **jcs:close()** function.

Parameters

- ***passphrase***—User's login password.
- ***remote-hostname***—Domain name or IP address of the remote router, switch, or security device. If you are opening a local connection, do not pass this value.
- ***username***—User's login name.

Return Value

- **connection**—Connection handle to the remote host.

Usage Examples

The following example shows how to connect to a local device:

```
var $connection = jcs:open();
```

The following example shows how to connect to a remote device:

```
var $connection = jcs:open(remote-hostname);
```

The following example shows how the user **bsmith** with a password **test123** obtains a connection handle to the server **fivestar**:

```
var $connection = jcs:open("fivestar", "bsmith", "test123");
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:close() Function on page 63**
- **jcs:execute() Function on page 65**

jcs:output() Function

Syntax

```
expr jcs:output('string');
```

```
<xsl:value-of select="jcs:output(string)"/>
```

Description

Generate unformatted output text that is immediately sent to the CLI session. In contrast, most script output is output at the end of the script.

Parameters

- ***string***—Text that is output immediately to the CLI session.

Usage Examples

SLAX syntax:

```
expr jcs:output('The VPN is up.');
```

XSLT syntax:

```
<xsl:value-of select="jcs:output('The VPN is up.')"/>
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:get-input() Function on page 67**

- jcs:get-secret( ) Function on page 68
- jcs:printf( ) Function on page 73
- jcs:progress( ) Function on page 74
- jcs:syslog( ) Function on page 78
- jcs:trace( ) Function on page 80

jcs:parse-ip( ) Function

Syntax

```
var $result = jcs:parse-ip("ipaddress/(prefix-length | netmask)");
<xsl:variable name="result" select="jcs:parse-ip('ipaddress/(prefix-length | netmask')"/>
```

Description Parse an IPv4 or IPv6 address.

Parameters

- ***ipaddress***—IPv4 or IPv6 address.
- ***prefix-length***—Prefix length defining the number of bits used in the network prefix portion of the address.
- ***netmask***—Netmask defining the network prefix portion of the address.

Return Value

- **result**—An array containing:
  - Host IP address (or **NULL** in the case of an error)
  - Protocol family (inet for IPv4 or inet6 for IPv6)
  - Prefix length
  - Network address
  - Network mask in dotted decimal notation (for IPv4 address; left blank for IPv6 addresses)

Usage Examples In the following examples, an IPv4 address and an IPv6 address are parsed and the resulting output is detailed:

```
var $addr = jcs:parse-ip("10.1.2.10/255.255.255.0");
```

- **$addr[1]** contains the host address **10.1.2.10**.
- **$addr[2]** contains the protocol family **inet**.
- **$addr[3]** contains the prefix length **24**.
- **$addr[4]** contains the network address **10.1.2.0**.
- **$addr[5]** contains the netmask for IPv4 **255.255.255.0**.

```
var $addr = jcs:parse-ip("080:0:0:0:8:800:200C:417A/100");
```

- **$addr[1]** contains the host address **80::8:800:200C:417A**.
- **$addr[2]** contains the protocol family **inet6**.
- **$addr[3]** contains the prefix length **100**.
- **$addr[4]** contains the network address **80::8:800:2000:0**.
- **$addr[5]** is blank for IPv6 (**""**).

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:break-lines( ) Function on page 63**
- **jcs:regex( ) Function on page 75**
- **jcs:split( ) Function on page 76**

jcs:printf( ) Function

Syntax

expr jcs:printf(*expression*);

<xsl:value-of select="jcs:printf(*expression*)"/>

Description

Generate formatted output text. Most standard **printf** formats are supported, in addition to some Junos OS–specific formats. The function returns a formatted string but does not print it on call. To use the following Junos OS modifiers, place the modifier between the percent sign (%) and the conversion specifier.

- **j1**—operator emits the field only if the field was changed from the last time the function was run. This assumes that the expression's format string is unchanged.
- **jc**—operator capitalizes the first letter of the associated output string.
- **jt{TAG}**—operator emits the tag if the field is not empty.

Parameters

- *expression*—format string containing an arbitrary number of format specifiers and associated arguments to output.

Usage Examples

For example:

```
<xsl:value-of select="jcs:printf('%-24j1s %-5jcs %-5jcs %s%jt{ - -> }s\n',
      'so-0/0/0', 'up', 'down', '10.1.2.3', '')"/>
<xsl:value-of select="jcs:printf('%-24j1s %-5jcs %-5jcs %s%jt{ - -> }s\n',
      'so-0/0/0', 'down', 'down', '10.1.2.3', '')"/>
```

produces the following output:

```
so-0/0/0         Up    Down  10.1.2.3
                 Down  Down  10.1.2.3
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60

- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:get-input() Function on page 67**
- **jcs:get-secret() Function on page 68**
- **jcs:output() Function on page 71**
- **jcs:progress() Function on page 74**
- **jcs:syslog() Function on page 78**
- **jcs:trace() Function on page 80**

jcs:progress() Function

Syntax

```
expr jcs:progress('string');
<xsl:value-of select="jcs:progress('string')"/>
```

Description Issue a progress message containing the single argument immediately to the CLI session provided that the **detail** flag was specified when the script was invoked.

Parameters

- ***string***—text output to CLI session

Usage Examples SLAX syntax:

```
expr jcs:progress('Working...');
```

XSLT syntax:

```
<xsl:value-of select="jcs:progress('Working...')"/>
```

The script must be invoked with the **detail** flag in order for the progress message to appear in the CLI session.

```
user@host> op script1.slax detail
```

```
2010-10-01 16:27:54 PDT: running op script 'script1.slax'
2010-10-01 16:27:54 PDT: opening op script '/var/db/scripts/op/script1.slax'
2010-10-01 16:27:54 PDT: reading op script 'script1.slax'
2010-10-01 16:27:54 PDT: Working...
2010-10-01 16:28:14 PDT: inspecting op output 'script1.slax'
2010-10-01 16:28:14 PDT: finished op script 'script1.slax'
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:get-input() Function on page 67**
- **jcs:get-secret() Function on page 68**
- **jcs:output() Function on page 71**
- **jcs:printf() Function on page 73**

- jcs:syslog( ) Function on page 78
- jcs:trace( ) Function on page 80

jcs:regex( ) Function

Syntax

```
var $result = jcs:regex(pattern, string);
<xsl:variable name="result" select="jcs:regex(pattern, string)"/>
```

Description Evaluate a regular expression against a given string argument and return any matches. This function requires two arguments: the regular expression and the string to which the regular expression is compared.

Parameters

- ***pattern***—Regular expression that is evaluated against the string argument.
- ***string***—String within which to search for matches of the specified regular expression.

Return Value

- **result**—Array of strings that match the given pattern within the string argument.

Usage Examples In the following example, the regex pattern consists of four distinct groups. The first group consists of the entire expression. The three subsequent groups are each of the parentheses–enclosed expressions within the main expression. The results for each **jcs:regex( )** function call contain an array of the matches.

```
var $pattern = "([0-9]+)(:*)([a-z]*)";
var $a = jcs:regex($pattern, "123:xyz");
var $b = jcs:regex($pattern, "r2d2");
var $c = jcs:regex($pattern, "test999!!!");

$a[1] == "123:xyz"   # string that matches the full reg expression
$a[2] == "123"       # ([0-9]+)
$a[3] == ":"         # (:*)
$a[4] == "xyz"       # ([a-z]*)
$b[1] == "2d"        # string that matches the full reg expression
$b[2] == "2"         # ([0-9]+)
$b[3] == ""          # (:*)  [empty match]
$b[4] == "d"         # ([a-z]*)
$c[1] == "999"       # string that matches the full reg expression
$c[2] == "999"       # ([0-9]+)
$c[3] == ""          # (:*)  [empty match]
$c[4] == ""          # ([a-z]*)  [empty match]
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- jcs:break-lines( ) Function on page 63
- jcs:parse-ip( ) Function on page 72
- jcs:split( ) Function on page 76

jcs:sleep( ) Function

Syntax

```
expr jcs:sleep(seconds, <milliseconds>);
<xsl:value-of select="jcs:sleep(seconds, milliseconds)"/>
```

Description Cause the script to sleep for a specified number of seconds and (optionally) milliseconds. You can use this function to help determine how a device component works over time. To do this, write a script that issues a command, calls the **jcs:sleep( )** function, and then reissues the same command.

Parameters

- ***milliseconds***—(Optional) Number of milliseconds the script should sleep.
- ***seconds***—Number of seconds the script should sleep.

Usage Examples In the following example, **jcs:sleep(1)** causes the script to sleep for 1 second, and **jcs:sleep(0, 10)** causes the script to sleep for 10 milliseconds.

SLAX syntax:

```
expr jcs:sleep(1);
expr jcs:sleep(0, 10);
```

XSLT syntax:

```
<xsl:value-of select="jcs:sleep(1)"/>
<xsl:value-of select="jcs:sleep(0, 10)"/>
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59

jcs:split( ) Function

Syntax

```
var $substrings = jcs:split(expression, string, <limit>);
<xsl:variable name="substrings" select="jcs:split(expression, string, limit)"/>
```

Description Split a string into an array of substrings delimited by a regular expression pattern. If the optional integer argument ***limit*** is specified, the function splits the entire string into ***limit*** number of substrings. If there are more than ***limit*** number of matches, the substrings include the first ***limit***-1 matches as well as the remaining portion of the original string for the last match.

Parameters

- ***expression***—Regular expression pattern used as the delimiter.
- ***limit***—(Optional) Number of substrings into which to break the original string.
- ***string***—Original string.

Return Value

- **$substrings**—Array of ***limit*** number of substrings

Usage Examples

In the following example, the original string is "123:abc:456:xyz:789". The jcs:split() function breaks this string into substrings that are delimited by the regular expression pattern, which in this case is a colon(:). The optional parameter ***limit*** is not specified, so the function returns an array containing all the substrings that are bounded by the delimiter(:).

```
var $pattern = "(:)";
var $substrings = jcs:split($pattern, "123:abc:456:xyz:789");
```

returns:

```
$substrings[1] == "123"
$substrings[2] == "abc"
$substrings[3] == "456"
$substrings[4] == "xyz"
$substrings[5] == "789"
```

The following example uses the same original string and regular expression as the previous example, but in this case, the optional parameter ***limit*** is included. Specifying ***limit***=2 causes the function to return an array containing only two substrings. The substrings include the first match, which is "**123**" (the same first match as in the previous example) and a second match, which is the remaining portion of the original string after the first occurrence of the delimiter.

```
var $pattern = "(:)";
var $substrings = jcs:split($pattern, "123:abc:456:xyz:789", 2);
```

returns:

```
$substrings[1] == "123"
$substrings[2] == "abc:456:xyz:789"
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:break-lines() Function on page 63**
- **jcs:parse-ip() Function on page 72**
- **jcs:regex() Function on page 75**

jcs:sysctl() Function

Syntax

```
var $value = jcs:sysctl(sysctl-value, "(i | s)");
```

```
<xsl:variable name="value" select="jcs:sysctl(sysctl-value, '(i | s)')"/>
```

Description

Return the value of the given **sysctl** value as a string or an integer. Use the "**i**" argument to specify an integer. Use the "**s**" argument to specify a string.

Parameters
- ***sysctl-value***—**sysctl** value to convert to a string or integer.

Return Value
- **$value**—Returned string or integer value.

Usage Examples

```
var $value = jcs:sysctl("kern.hostname", "s");
```

Related Documentation
- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59

jcs:syslog( ) Function

Syntax

```
expr jcs:syslog(priority, message, <message2>, <message3> ...);
<xsl:value-of select="jcs:syslog(priority, messsage <message2>, <message3>)"/>
```

Description

Log messages with the specified priority to the system log file. The priority can be expressed as a ***facility.severity*** string or as a calculated integer. The ***message*** argument is a string or variable that is written to the system log file. Optionally, additional strings or variables can be included in the argument list. The ***message*** argument is concatenated with any additional parameters, and the concatenated string is written to the system log file. The syslog file is specified at the **[edit system syslog]** hierarchy level of the device configuration file.

Parameters
- ***message***—String or variable that is output to the system log file.
- ***message2***—(Optional) Any additional number of strings or variable names passed as arguments to the function. These are concatenated with the ***message*** argument and output to the syslog.
- ***priority***—Priority given to the syslog message. The priority can be specified as a ***facility.severity*** string, or it can expressed as an integer calculated from the corresponding numeric values of the facility and severity strings. Table 8 on page 78 and Table 9 on page 79 show the facility and severity strings available and their corresponding numeric values.

  The integer value of the ***priority*** parameter is calculated by multiplying the facility string numeric value by 8 and adding the severity string numeric value. For example, if the ***facility.severity*** string pair is **"pfe.alert"**, the priority value is 161 ((20 x 8)+1).

Table 8: Facility Strings

Facility String	Description	Numeric Value
auth	Authorization system	4
change	Configuration change log	22
conflict	Configuration conflict log	21
daemon	Various system processes	3

Table 8: Facility Strings *(continued)*

Facility String	Description	Numeric Value
external	Local external applications	18
firewall	Firewall filtering system	19
ftp	FTP processes	11
interact	Commands executed by the UI	23
pfe	Packet Forwarding Engine	20
user	User processes	1

Table 9: Severity Strings

Severity String	Description	Numeric Value
alert	Conditions that should be corrected immediately	1
crit	Critical conditions	2
debug	Debug messages	7
emerg or **panic**	Panic conditions	0
err or **error**	Error conditions	3
info	Informational messages	6
notice	Conditions that should be specially handled	5
warn or **warning**	Warning messages	4

Usage Examples

The following three examples log **pfe** messages with an **alert** priority. The string "**mymessage**" is output to the system log file. All three examples are equivalent:

```
expr jcs:syslog("pfe.alert", "mymessage");

expr jcs:syslog(161, "mymessage");

var $message = "mymessage";
expr jcs:syslog("pfe.alert", $message);
```

The following example logs **pfe** messages with an **alert** priority similar to the previous example. In this example, however, there are additional string parameters. For this case, the concatenated string "**mymessage mymessage2**" is output to the system log file.

```
expr jcs:syslog("pfe.alert", "mymessage ", "mymessage2");
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:get-input( ) Function on page 67**
- **jcs:get-secret( ) Function on page 68**
- **jcs:output( ) Function on page 71**
- **jcs:printf( ) Function on page 73**
- **jcs:progress( ) Function on page 74**
- **jcs:trace( ) Function on page 80**

jcs:trace( ) Function

Syntax

```
expr jcs:trace('expression');
<xsl:value-of select="jcs:trace('expression')"/>
```

Description Issue a trace message, which is sent to the trace file.

Parameters

- ***expression***—String that is output to the trace file.

Usage Examples SLAX syntax:

```
expr jcs:trace('test');
```

XSLT syntax:

```
<xsl:value-of select="jcs:trace('test')"/>
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- **jcs:get-input( ) Function on page 67**
- **jcs:get-secret( ) Function on page 68**
- **jcs:output( ) Function on page 71**
- **jcs:printf( ) Function on page 73**
- **jcs:progress( ) Function on page 74**
- **jcs:syslog( ) Function on page 78**

Junos Script Automation: Named Templates in the jcs Namespace Overview

Named templates are provided in the junos.xsl import file to make scripting tasks easier in commit, op, and event scripts. Named templates have the **jcs:** prefix to indicate they

belong to the **jcs:** namespace. The templates use the **jcs:** prefix to avoid conflicting with user-defined templates of the same name in a script.

When you use named templates in a script, you must map to the **jcs** namespace in your style sheet declaration by including the **xmlns:jcs** attribute in the opening **<xsl:stylesheet>** tag element. You must also include the **<xsl:import/>** tag element or **import** statement to import the junos.xsl file. Both of these steps are shown in the following examples:

XSLT Syntax

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
      xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>
  ...
</xsl: stylesheet>
```

SLAX Syntax

```
version 1.0;
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";
```

You then include an **<xsl:call-template name="*name*">** element in the script to reference each named template, passing in any required or optional parameters.

For more information about attributes and tag elements to include in your scripts, see “Required Boilerplate for Commit Scripts” on page 133, “Required Boilerplate for Op Scripts” on page 307, and “Required Boilerplate for Event Scripts” on page 439.

Related Documentation

- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- Junos Script Automation: Predefined Parameters in the junos.xsl File on page 92

Junos Named Templates in the jcs Namespace Summary

The Junos named templates are summarized in the following table:

Table 10: Junos Named Templates

Template	Description
jcs:edit-path	Generate an **<edit-path>** element suitable for inclusion in an **<xnm:error>** or **<xnm:warning>** element.
jcs:emit-change	Generate a **<change>** or **<transient-change>** element, which results in a persistent or transient change to the configuration.
jcs:emit-comment	Emit a simple comment that indicates a change was made by a commit script.
jcs:load-configuration	Make structured changes to the Junos OS configuration using an op script.
jcs:statement	Generate a **<statement>** element suitable for inclusion in an **<xnm:error>** or **<xnm:warning>** element.

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 60
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- Junos Script Automation: Named Templates in the jcs Namespace Overview on page 80
- Junos Script Automation: Predefined Parameters in the junos.xsl File on page 92

Junos Named Templates in the jcs Namespace

The templates are discussed in more detail in the following sections:

- **jcs:edit-path Template on page 82**
- **jcs:emit-change Template on page 83**
- **jcs:emit-comment Template on page 85**
- **jcs:load-configuration Template on page 86**
- **jcs:statement Template on page 89**

jcs:edit-path Template

Syntax

```
<xsl:call-template name="jcs:edit-path">
  <xsl:with-param name="dot" select="expression"/>
</xsl:call-template>
```

Description Generate an **<edit-path>** element suitable for inclusion in an **<xnm:error>** or **<xnm:warning>** element. By default, the location of the configuration error is passed as **dot** into the **<jcs:edit-path>** template. This location defaults to " **.** " , the current position in the XML hierarchy. You can alter the default by including the **select** attribute of the **dot** parameter.

Parameters **<xsl:param name="dot" select=".">**—Allows you to indicate a location other than the current location in the XML hierarchy. The **select** attribute contains the current context " **.** " as a default value. If you want to change the current context, you can include the **dot** parameter and include a different XPath expression in the **select** attribute.

Usage Examples The following example demonstrates how to call this template in a commit script and set the context to the **[edit chassis]** hierarchy level:

The **<jcs:edit>** template generates an **<edit-path>** element suitable for inclusion in an **<xnm:error>** or **<xnm:warning>** element. The location of the configuration error is passed as **dot** into the **<jcs:edit-path>** template. This location defaults to " **.** " , the current position in the XML hierarchy. You can alter the default by including the **select** attribute of the **dot** parameter. The following example demonstrates how to call this template in a commit script and set the context to the **[edit chassis]** hierarchy level:

```
<xsl:if test="not(chassis/source-route)">
  <xnm:warning>
    <xsl:call-template name="jcs:edit-path">
      <xsl:with-param name="dot" select="chassis"/>
    </xsl:call-template>
```

```
      <message>IP source-route processing is not enabled.</message>
    </xnm:warning>
  </xsl:if>
```

When you commit a configuration that does not enable IP source routing, the **<xnm:warning>** element results in the following command-line interface (CLI) output:

```
user@host# commit
[edit chassis] # The hierarchy level is generated by the <jcs:edit-path> template.
    warning: IP source-route processing is not enabled.
commit complete
```

Related Documentation

- Example: Requiring and Restricting Configuration Statements on page 201
- Junos Named Templates in the jcs Namespace Summary on page 81
- **xsl:param on page 104**
- **xsl:with-param on page 108**

jcs:emit-change Template

Syntax

```
<xsl:call-template name="jcs:emit-change">
  <xsl:with-param name="content">
    ...
  </xsl:with-param>
  <xsl:with-param name="dot" select="expression"/>
  <xsl:with-param name="message">
    <xsl:text>...</xsl:text>
  </xsl:with-param>
  <xsl:with-param name="name" select="name($dot)"/>
  <xsl:with-param name="tag" select="'change'"/>
  <xsl:with-param name="tag" select="'transient-change'"/>
</xsl:call-template>
```

Description Generate a **<change>** or **<transient-change>** element, which results in a persistent or transient change to the configuration.

Parameters This template includes the following optional parameters:

- **<xsl:param name="content">**—Allows you to include the content of the change, relative to **dot**.
- **<xsl:param name="dot" select=".">**—Allows you to indicate a location other than the current location in the XML hierarchy. The **select** attribute contains the current context "." as a default value. If you want to change the current context, you can include the **dot** parameter and include a different XPath expression in the **select** attribute.
- **<xsl:param name="message">**—Allows you to include a warning message to be displayed by the CLI, notifying the user that the configuration has been changed. The message parameter automatically includes the edit path, which defaults to the current location in the XML hierarchy. To change the default edit path, include the **dot** parameter.

- **<xsl:param name="name" select="name($dot)"/>**—Allows you to refer to the current element or attribute. The **name()** XPath function returns the name of an element or attribute. The **name** parameter defaults to the name of the element in **$dot** (which in turn defaults to "**.**", which is the current element).
- **<xsl:param name="tag" select="'change'"/>**—Allows you to specify the type of change to be generated. By default, the **<jcs:emit-change>** template generates a permanent change, as designated by the **'change'** expression. To specify a transient change, you must include the **tag** parameter and include the **'transient-change'** expression, as shown here:

```
<xsl:with-param name="tag" select="'transient-change'"/>
```

Usage Examples The following example demonstrates how to call this template in a commit script:

```
<xsl:template match="configuration">
  <xsl:for-each select="interfaces/interface/unit[family/iso]">
    <xsl:if test="not(family/mpls)">
      <xsl:call-template name="jcs:emit-change">
        <xsl:with-param name="message">
          <xsl:text>Adding 'family mpls' to ISO-enabled interface</xsl:text>
        </xsl:with-param>
        <xsl:with-param name="content">
          <family>
            <mpls/>
          </family>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:if>
  </xsl:for-each>
</xsl:template>
```

When you commit a configuration that includes one or more interfaces that have IS-IS enabled but do not have the **family mpls** statement included at the **[edit interfaces** *interface-name* **unit** *logical-unit-number***]** hierarchy level, the **<jcs:emit-change>** template adds the **family mpls** statement to the configuration and generates the following CLI output:

```
[edit]

user@host# commit
[edit interfaces interface so-1/2/3 unit 0]
    warning: Adding 'family mpls' to ISO-enabled interface
[edit interfaces interface so-1/2/3 unit 0]
    warning: Adding ISO-enabled interface so-1/2/3.0 to [protocols mpls]
[edit interfaces interface so-1/3/2 unit 0]
    warning: Adding 'family mpls' to ISO-enabled interface
[edit interfaces interface so-1/3/2 unit 0]
    warning: Adding ISO-enabled interface so-1/3/2.0 to [protocols mpls]
commit complete
```

The **content** parameter of the **<jcs:emit-change>** template provides a simpler method for specifying a change to the configuration. For example, consider the following code:

```
<xsl:with-param name="content">
  <family>
    <mpls/>
```

```
    </family>
  </xsl:with-param>
```

The **<jcs:emit-change>** template converts the **content** parameter into a **<change>** request. The **<change>** request inserts the provided partial configuration content into the complete hierarchy of the current context node. Thus, the **<jcs:emit-change>** template changes the hierarchy information in the **content** parameter into the following code:

```
<change>
  <interfaces>
    <interface>
      <name><xsl:value-of select="name"/></name>
      <unit>
        <name><xsl:value-of select="unit/name"/></name>
        <family>
          <mpls/>
        </family>
      </unit>
    </interface>
  </interfaces>
</change>
```

If a transient change is required, the **tag** parameter can be passed in as '**transient-change**', as shown here:

```
<xsl:with-param name="tag" select="'transient-change'"/>
```

The extra quotation marks are required to allow XSLT to distinguish between the string "**transient-change**" and the contents of a node named "**transient-change**". If the change is relative to a node other than the context node, the parameter "**dot**" can be set to that node, as shown in the following example, where context is set to the **[edit chassis]** hierarchy level:

```
<xsl:for-each select="interfaces/interface/unit">
  ...
  <xsl:call-template name="jcs:emit-change">
    <xsl:with-param name="dot" select="chassis"/>
      ...
```

Related Documentation

- Example: Imposing a Minimum MTU Setting on page 207.
- Junos Named Templates in the jcs Namespace Summary on page 81
- **xsl:param on page 104**
- **xsl:with-param on page 108**

jcs:emit-comment Template

Syntax

```
<junos:comment>
  <xsl:text>...</xsl:text>
</junos:comment>
```

Description

Emit a simple comment that indicates a change was made by a commit script. The template contains a **<junos:comment>** element. You never call the <**jcs:emit-comment**> template directly. Rather, you include its **<junos:comment>** element and the child element

<xsl:text> inside a call to the **<jcs:emit-change>** template, a **<change>** element, or a **<transient-change>** element.

Usage Examples The following example demonstrates how to call this template in a commit script:

```
<xsl:call-template name="jcs:emit-change">
  <xsl:with-param name="content">
    <term>
      <name>very-last</name>
      <junos:comment>
        <xsl:text>This term was added by a commit script</xsl:text>
      </junos:comment>
      <then>
        <accept/>
      </then>
    </term>
  </xsl:with-param>
</xsl:call-template>
```

When you issue the **show firewall** configuration mode command, the following output appears:

```
[edit]

user@host# show firewall
family inet {
    term very-last {
          /* This term was added by a commit script */
    then accept;
    }
}
```

Related Documentation

- Example: Adding a Final then accept Term to a Firewall on page 234.
- Junos Named Templates in the jcs Namespace Summary on page 81
- **jcs:emit-change Template on page 83**

jcs:load-configuration Template

Syntax

```
<xsl:call-template name="jcs:load-configuration">
  <xsl:with-param name="action" select="(merge | override | replace)"/>
  <xsl:with-param name="commit-options" select="node-set"/>
  <xsl:with-param name="configuration" select="configuration-data"/>
  <xsl:with-param name="connection" select="connection-handle"/>
</xsl:call-template>
```

Description Make structured changes to the Junos OS configuration using an op script. When called, the template locks the configuration database, loads the configuration changes, commits the configuration, and then unlocks the configuration database.

The **jcs:load-configuration** template makes changes to the configuration in **configure exclusive** mode. In this mode, Junos OS locks the candidate *global* configuration for as long as the script accesses the shared database and makes changes to the configuration without interference from other users.

If another user is currently editing the configuration in **configure exclusive** mode or if the database is already locked when the template is called, the call fails. In addition, if there are existing, uncommitted changes to the configuration when the template is called, the commit will fail. If the template call is successful but the commit fails, Junos OS discards the uncommitted changes and rolls back the configuration.

Parameters

- **action**—Specifies how to load the configuration changes with respect to the candidate configuration. The following options are supported:
  - **merge**—Combine the candidate configuration and the incoming configuration changes. If the candidate configuration and the incoming configuration contain conflicting statements, the incoming statements override those in the candidate configuration.
  - **override**—Replace the entire candidate configuration.
  - **replace**—Replace existing statements in the candidate configuration with the tags of the same name that are marked with **replace:** in the incoming configuration. If there is no existing statement of the same name in the candidate configuration, the statement is added to the candidate configuration.
- **commit-options**—Node-set defining options that customize the commit command. The default value is null. Supported commit options are:
  - **check**—Check the correctness of the candidate configuration syntax, but do not commit the changes.
  - **force-synchronize**—Force the commit on the other Routing Engine (ignore any warnings).
  - **log**—Write the specified message to the commit log. This is identical to the CLI configuration mode command **commit comment**.
  - **synchronize**—Synchronize the commit on both Routing Engines.
- **configuration**—XML configuration changes.
- **connection**—Connection handle generated by a call to the **jcs:open( )** function.

Usage Examples

The following example calls the **jcs:load-configuration** template to modify the configuration to disable an interface. The interface name is supplied by the user and stored in the variable **$interface-name**. All of the values required for the **jcs:load-configuration** template are defined as variables, which are then passed into the template as arguments.

In this example, the configuration data that includes the changes to the configuration are stored in the variable **$disable**. This is the value used for the **$configuration** parameter of the **jcs:load-configuration** template. The **$load-action** variable is initialized to **merge**, which merges the configuration changes in the **$disable** variable with the candidate configuration. This is the equivalent of the CLI configuration mode command **load merge**.

The **$options** variable uses the **:=** operator to create a node-set, which is passed to the template as the value of the **$commit-options** parameter. This example uses the

synchronize commit option . If the commit succeeds, it will commit the configuration changes on both Routing Engines. The **log** tag is also included to add the description of the commit to the commit log file for future reference.

The call to the **jcs:open()** function opens a connection with the Junos OS management process (mgd) and returns a connection handle that is stored in the **$conn** variable. All of the defined variables are passed as arguments to the **jcs:load-configuration** template at the time that it is called.

SLAX syntax:

```
var $disable = {
  <configuration> {
    <interfaces> {
      <interface> {
        <name> $interface-name;
        <disable>;
      }
    }
  }
}
var $load-action = "merge";
var $options := {
  <commit-options> {
    <synchronize>;
    <log> "disabling interface on both routing engines";
}
var $conn = jcs:open();

var $disable-results := {
  call jcs:load-configuration($action=$load-action, $commit-options=$options,
      $configuration = $disable, $connection = $conn);
}
if ($disable-results//xnm:error) {
  for-each ($disable-results//xnm:error) {
    <output> message;
  }
}
var $close-results = jcs:close($conn);
```

The **:=** operator copies the results of the **jcs:load-configuration** template call to a temporary variable and runs the **node-set** function on that variable. The **:=** operator ensures that the **$disable-results** variable is a node-set rather than a result tree fragment so that the script can access the contents. The **if** code block is included to output any error messages that may indicate a problem in committing the configuration. The **jcs:close** function closes the connection.

In XSLT, the code corresponding to the SLAX call to **jcs:load-configuration** template is:

```
<xsl:variable name="disable-results-temp">
  <xsl:call-template name="jcs:load-configuration">
    <xsl:with-param name="action" select="$load-action"/>
    <xsl:with-param name="commit-options" select="$options"/>
    <xsl:with-param name="configuration" select="$disable"/>
    <xsl:with-param name="connection" select="$conn"/>
```

```
  </xsl:call-template>
</xsl:variable>

<xsl:variable xmlns ext="http:xmlsoft.org/XSLT/namespace" \
  name="disable-results" select="ext:node-set($disable-results-temp)"/>
```

Related Documentation

- Changing the Configuration Using Op Scripts on page 329
- Junos Named Templates in the jcs Namespace Summary on page 81
- SLAX Operators on page 55
- **jcs:close( ) Function on page 63**
- **jcs:open( ) Function on page 70**

jcs:statement Template

Syntax

```
<xsl:call-template name="jcs:statement">
  <xsl:with-param name="dot" select="expression"/>
</xsl:call-template>
```

Description Generate a **<statement>** element suitable for inclusion in an **<xnm:error>** or **<xnm:warning>** element. The parameter **dot** can be passed into the **<jcs:statement>** template if the error is not at the current position in the XML hierarchy.

Parameters **<xsl:param name="dot" select=".">**—Allows you to indicate a location other than the current location in the XML hierarchy. The **select** attribute contains the current context " **.**" as a default value. If you want to change the current context, you can include the **dot** parameter and include a different XPath expression in the **select** attribute.

Usage Examples The following example demonstrates how to call this template in a commit script:

```
<xnm:error>
  <xsl:call-template name="jcs:edit-path"/>
  <xsl:call-template name="jcs:statement">
    <xsl:with-param name="dot" select="mtu"/>
  </xsl:call-template>
  <message>
    <xsl:text>SONET interfaces must have a minimum MTU of </xsl:text>
    <xsl:value-of select="$min-mtu"/>
    <xsl:text>.</xsl:text>
  </message>
</xnm:error>
```

When you commit a configuration that includes a SONET/SDH interface with a maximum transmission unit (MTU) setting less than a specified minimum, the **<xnm:error>** element results in the following CLI output:

```
[edit]

user@host# commit
[edit interfaces interface so-1/2/3]
    'mtu 576;'  # mtu statement generated by the <jcs:statement> template
    SONET interfaces must have a minimum MTU of 2048.
error: 1 error reported by commit scripts
```

```
error: commit script failure
```

The test of the MTU setting is not performed in the **<xnm:error>** element. For the full example, see “Example: Imposing a Minimum MTU Setting” on page 207.

Related Documentation

- Example: Configuring Dual Routing Engines on page 252.
- Junos Named Templates in the jcs Namespace Summary on page 81
- **xsl:param on page 104**
- **xsl:with-param on page 108**

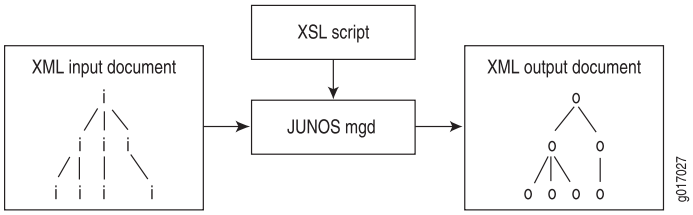
xsl:template match="/" Template

Syntax <xsl:template match="/">

Description The **<xsl:template match="/">** template is an unnamed template in the junos.xsl file that allows you to use shortened XPath expressions in your scripts. You must import the junos.xsl file to use this template. However, because this template is not in the **jcs** namespace, you do not need to map to the **jcs** namespace in your style sheet declaration in order to use this template.

The Junos OS management process (mgd) generates the output document as the product of its evaluation of the input document, as shown in Figure 3 on page 90.

Figure 3: Commit Script Input and Output



Generally, an XSLT engine uses recursion to evaluate the entire input document. However, the **<xsl:apply-templates>** instruction allows you to limit the scope of the evaluation so that the management process (the Junos OS's XSLT engine) must evaluate only a subset of the input document.

The **<xsl:template match="/">** template is an unnamed template that uses the **<xsl:apply-templates>** instruction to specify the contents of the input document's **<configuration>** element as the only node to be evaluated in the generation of the output document.

The **<xsl:template match="/">** template contains the following tags:

```
1 <xsl:template match="/">
2   <commit-script-results>
3     <xsl:apply-templates select="commit-script-input/configuration"/>
4   </commit-script-results>
5 </xsl:template>
```


Line 1 matches the root node of the input document. When the management process sees the root node of the input document, this template is applied.

```
1  <xsl:template match="/">
```

Line 2 designates the root, top-level tag of the output document. Thus, Line 2 specifies that the evaluation of the input document results in an output document whose top-level tag is **<commit-script-results>**.

```
2     <commit-script-results>
```

Line 3 limits the scope of the evaluation of the input document to the contents of the **<configuration>** element, which is a child of the **<commit-script-input>** element.

```
3        <xsl:apply-templates select="commit-script-input/configuration"/>
```

Lines 4 and 5 are closing tags.

You do not need to explicitly include the **<xsl:template match="/">** template in your scripts because this template is included in the import file junos.xsl.

When the **<xsl:template match="/">** template executes the **<xsl:apply-templates>** instruction, the script jumps to a template that matches the **<configuration>** tag. This template, **<xsl:template match="configuration">**, is part of the commit script boilerplate that you must include in all of your commit scripts:

```
<xsl:template match="configuration">
  <!-- ... insert your code here ... -->
</xsl:template>
```

Thus, the import file junos.xsl contains a template that points to a template explicitly referenced in your script.

Usage Examples

The following example contains the **<xsl:if>** programming instruction and the **<xnm:warning>** element. The logical result of both templates is:

```
<commit-script-results>   <!-- from template in junos.xsl import file -->
  <xsl:if test="not(system/host-name)"> <!-- from "configuration" template -->
    <xnm:warning xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
      <edit-path>[edit system]</edit-path>
      <statement>host-name</statement>
      <message>Missing a hostname for this device.</message>
    </xnm:warning>
  </xsl:if>   <!-- end of "configuration" template -->
</commit-script-results>   <!-- end of template in junos.xsl import file -->
```

When you import the junos.xsl file and explicitly include the **<xsl:template match="configuration">** tag in your commit script, the context (**dot**) moves to the **<configuration>** node. This allows you to write all XPath expressions relative to that point. This technique allows you to simplify the XPath expressions you use in your commit scripts. For example, instead of writing this, which matches the device with hostname **atlanta**:

```
<xsl:if test="starts-with(commit-script-input/configuration/system/host-name,
'atlanta')">
```

You can write this:

```
<xsl:if test="starts-with(system/host-name, 'atlanta')">
```

Related Documentation

- Junos Named Templates in the jcs Namespace Summary on page 81

Junos Script Automation: Predefined Parameters in the junos.xsl File

There are several predefined parameters, which are declared within the junos.xsl file. These six standard parameters provide valuable information about the Junos OS environment and are available to any script that imports the file. Table 11 on page 92 describes the built-in arguments

Table 11: Predefined Parameters Available to Automation Scripts

Name	Description	Example
$hostname	Host name of the local device	Tokyo
$localtime	Local time when the script was executed	Mon Apr 26 13:15:25 2010
$localtime_iso	Local time when the script was executed, in ISO format	2010-04-26 13:15:25 PDT
$product	Model of the local device	m10i
$script	Filename of script being executed	test.slax
$user	User name	root

To use the predefined parameters in a script, you must include the **<xsl:import>** tag element or **import** statement to import the junos.xsl file. This is shown in the following examples:

XSLT Syntax

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0">
  <xsl:import href="../import/junos.xsl"/>
  ...
</xsl: stylesheet>
```

SLAX Syntax

```
version 1.0;
import "../import/junos.xsl";
```

You can access the standard arguments in your script by including the XSLT **xsl:param** tag or the SLAX **param** statement at the global level of the script.

XSLT:

```
<xsl:param name="product"/>
```

SLAX:

```
param $product;
```

You can then access the parameters throughout the script as you would access any variable.

Related Documentation

- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 59
- Junos Script Automation: Named Templates in the jcs Namespace Overview on page 80
- SLAX Variables Overview on page 50
- **var on page 120**

CHAPTER 7

Summary of XPath and XSLT Constructs

This chapter discusses the following topics:

Summary of Standard XPath and XSLT Functions Referenced in This Guide

Junos OS commit scripts, op scripts, and event scripts support all functions defined in the Extensible Markup Language Path Language (XPath) and Extensible Stylesheet Language Transformations (XSLT) scripting languages. This section describes only the functions referenced in this guide. The functions are organized alphabetically.

concat()

Syntax *string* concat(*string*, *string*+)

Description Return the concatenation of the arguments.

Usage Examples See "Example: Limiting the Number of E1 Interfaces" on page 209, "Example: Controlling IS-IS and MPLS Interfaces" on page 221, "Example: Adding T1 Interfaces to a RIP Group" on page 224, "Example: Configuring Administrative Groups for LSPs" on page 248, and "Example: Configuring Dual Routing Engines" on page 252.

contains()

Syntax *boolean* contains(*string*, *string*)

Description Return TRUE if the first argument string contains the second argument string, and otherwise returns FALSE.

Usage Examples See "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

Related Documentation
- **starts-with() on page 97**

count()

Syntax *number* count(*node-set*)

Description Return the number of nodes in the argument node-set.

Usage Examples See "Example: Limiting the Number of E1 Interfaces" on page 209.

Related Documentation
- **last() on page 96**
- **position() on page 97**

last()

Syntax *number* last()

Description Return the index of the last node in the list that is currently being evaluated.

Usage Examples See "Example: Limiting the Number of E1 Interfaces" on page 209.

Related Documentation
- **count() on page 96**
- **position() on page 97**

name()

Syntax *string* name(*<node-set>*)

Description Return the full name of the last node in the node set, including the prefix for its namespace declared in the source document. If no argument is passed, then returns the full name of the context node.

Usage Examples See **jcs:emit-change Template**.

not()

Syntax *boolean* not(*boolean*)

Description Return TRUE if its argument is FALSE, and FALSE if the argument is TRUE.

Usage Examples See "Example: Requiring and Restricting Configuration Statements" on page 201, "Example: Controlling IS-IS and MPLS Interfaces" on page 221, "Example: Configuring a Default Encapsulation Type" on page 227, "Example: Controlling LDP Configuration" on page 230, "Example: Adding a Final then accept Term to a Firewall" on page 234, "Example: Configuring Administrative Groups for LSPs" on page 248, "Example: Configuring Dual Routing Engines" on page 252, and "Example: Preventing Import of the Full Routing Table" on page 256.

position()

Syntax *number* position()

Description Return the position of the context node among the list of nodes that are currently being evaluated.

Usage Examples See "Example: Adding a Final then accept Term to a Firewall" on page 234 and "Example: Prepending a Global Policy" on page 265.

Related Documentation
- **count() on page 96**
- **last() on page 96**

starts-with()

Syntax *boolean* starts-with(*string*, *string*)

Description Return TRUE if the first argument string starts with the second argument string, and returns FALSE otherwise.

Usage Examples See "Example: Imposing a Minimum MTU Setting" on page 207, "Example: Limiting the Number of E1 Interfaces" on page 209, "Example: Limiting the Number of ATM Virtual Circuits" on page 217, "Example: Adding T1 Interfaces to a RIP Group" on page 224, "Example: Configuring a Default Encapsulation Type" on page 227, and "Example: Configuring Dual Routing Engines" on page 252.

Related Documentation
- **contains() on page 96**

string-length()

Syntax *number* string-length(*<string>*)

Description Return the number of characters in the string. If the argument is omitted, it returns the string value of the context node.

Usage Examples See "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

substring-after()

Syntax *string* substring-after(*string*, *string*)

Description Return the substring of the first argument string that occurs after the second argument string. If the second string is not contained in the first string, or if the second string is empty, then it returns an empty string.

Usage Examples See "Example: Limiting the Number of E1 Interfaces" on page 209 and "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

Related Documentation
- **substring-before() on page 98**

substring-before()

Syntax *string* substring-before(*string*, *string*)

Description Return the substring of the first argument string that occurs before the second argument string. If the second string is not contained in the first string, or if the second string is empty, then it returns an empty string.

Usage Examples See "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

Related Documentation
- **substring-after() on page 98**

Summary of Standard XSLT Elements and Attributes Referenced in This Guide

Junos OS commit scripts, op scripts, and event scripts support all elements and attributes defined in the Extensible Markup Language Path Language (XPath) and Extensible Stylesheet Language Transformations (XSLT) scripting languages. This section describes only the elements and attributes referenced in this guide. The elements and attributes are organized alphabetically.

- xsl:stylesheet on page 104
- xsl:template on page 105
- xsl:text on page 106
- xsl:value-of on page 106
- xsl:variable on page 107
- xsl:when on page 107
- xsl:with-param on page 108

xsl:apply-templates

Syntax

```
<xsl:apply-templates select="node-set-expression">
  <xsl:with-param name="qualified-name" select="expression">
    ...
  </xsl:with-param>
</xsl:apply-templates>
```

Description Apply one or more templates, according to the value of the **select** attribute. The nodes to which the processor applies templates are selected by the path specified by the **select** attribute. The **<xsl:template>** instruction dictates which elements are transformed according to which template. The templates that are applied are passed the parameters specified by the **<xsl:with-param>** elements within the **<xsl:apply-templates>** instruction.

Attributes **select**—Selects the nodes to which the processor applies templates. By default, the processor applies templates to the child nodes of the current node.

Usage Examples See "Example: Adding a Final then accept Term to a Firewall" on page 234 and "Example: Preventing Import of the Full Routing Table" on page 256.

Related Documentation

- xsl:call-template on page 100
- xsl:for-each on page 102
- xsl:template on page 105
- xsl:with-param on page 108

xsl:call-template

Syntax

```
<xsl:call-template name="qualified-name">
  <xsl:with-param name="qualified-name" select="expression">
    ...
  </xsl:with-param>
</xsl:call-template>
```

Description Call a named template. The **<xsl:with-param>** elements within the **<xsl:call-template>** instruction are used to define parameters that are passed to the template.

Attributes **name**—Specifies the name of the called template.

Usage Examples See "Example: Requiring and Restricting Configuration Statements" on page 201, "Example: Imposing a Minimum MTU Setting" on page 207, and "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

Related Documentation

- **xsl:apply-templates on page 99**
- **xsl:template on page 105**

xsl:choose

Syntax

```
<xsl:choose>
  <xsl:when test="boolean-expression">
  ...
  </xsl:when>
  <xsl:otherwise>
  ...
  </xsl:otherwise>
</xsl:call-template>
```

Description Express multiple conditional tests. The **<xsl:choose>** instruction contains one or more **<xsl:when>** elements, each of which tests an XPath expression. If the test evaluates as TRUE, the XSLT processor executes the instructions in the **<xsl:when>** element. The XSLT processor processes only the instructions contained in the first **<xsl:when>** element whose **test** attribute evaluates as TRUE. If none of the **<xsl:when>** elements' **test** attributes evaluate as TRUE, the content of the **<xsl:otherwise>** element, if there is one, is processed.

Usage Examples See "Example: Configuring Dual Routing Engines" on page 252, "Example: Preventing Import of the Full Routing Table" on page 256, and "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

Related Documentation

- **xsl:if on page 102**
- **xsl:otherwise on page 103**
- **xsl:when on page 107**

xsl:comment

Syntax

```
<xsl:comment>
  ...
</xsl:comment>
```

Description Generate a comment within the final document. The content within the **<xsl:comment>** element determines the value of the comment. The content must not contain two hyphens next to each other (**- -**); this sequence is not allowed in comments.

XSLT files can contain ordinary **<!-- ... Insert your comment here ... -->** comments, but these are ignored by the processor. To generate a comment within the final document, use an **<xsl:comment>** element.

Usage Examples See "Example: Adding a Final then accept Term to a Firewall" on page 234.

xsl:copy-of

Syntax

```
<xsl:copy-of select="expression"/>
```

Description Create a copy of what is selected by the expression defined in the **select** attribute. Namespace nodes, child nodes, and attributes of the current node are automatically copied as well.

Attributes **select**—Specifies an expression to select nodes to be copied.

Usage Examples See "Example: Requiring and Restricting Configuration Statements" on page 201.

Related Documentation

- **xsl:value-of on page 106**

xsl:element

Syntax

```
<xsl:element name="expression"/>
```

Description Create an element node in the output document.

Attributes **name**—Specifies the name of the element to be created. The value of the **name** attribute can be set to an expression that is extracted from the input XML document. To do this, enclose an XML element in curly brackets (**{ }**), as in **<xsl:element name="{$isis-level-1}"**.

Usage Examples See "Example: Creating a Complex Configuration Based on a Simple Interface Configuration" on page 242.

xsl:for-each

Syntax

```
<xsl:for-each select="node-set-expression">
  ...
</xsl:for-each>
```

Description Include a looping mechanism that repeats XSL processing for each instance of identical XML elements. The element nodes are selected by the expression defined by the **select** attribute. Each of the nodes is then processed by the instructions contained in the **<xsl:for-each>** instruction.

Attributes **select**—Specifies an expression to select nodes to be processed.

Usage Examples See "Example: Requiring and Restricting Configuration Statements" on page 201, "Example: Imposing a Minimum MTU Setting" on page 207, "Example: Limiting the Number of E1 Interfaces" on page 209, "Example: Adding T1 Interfaces to a RIP Group" on page 224, "Example: Configuring Administrative Groups for LSPs" on page 248, and "Example: Configuring Dual Routing Engines" on page 252.

Related Documentation

- **xsl:apply-templates on page 99**

xsl:if

Syntax

```
<xsl:if test="boolean-expression">
  ...
</xsl:if>
```

Description Include a conditional construct that causes instructions to be processed if the Boolean expression held in the **test** attribute evaluates to TRUE.

Attributes **test**—Specifies a Boolean expression.

Usage Examples See "Example: Requiring and Restricting Configuration Statements" on page 201, "Example: Limiting the Number of E1 Interfaces" on page 209, "Example: Adding T1 Interfaces to a RIP Group" on page 224, and "Example: Configuring Dual Routing Engines" on page 252.

Related Documentation

- **xsl:choose on page 100**
- **xsl:when on page 107**

xsl:import

Syntax

```
<xsl:import href="../import/junos.xsl"/>
```

Description Import rules from an external style sheet. Provides access to all the declarations and templates within the imported style sheet, and allows you to override them with your own if needed. Any **<xsl:import>** elements must be the first elements within the style sheet, the first children of the **<xsl:stylesheet>** document element. The path can be any URI. The **../import/junos.xsl** path shown in the syntax is standard for all commit scripts, op scripts, and event scripts.

Imported rules are overwritten by any subsequent matching rules within the importing style sheet. If more than one style sheet is imported, the style sheets imported last override each previous import where the rules match.

Attributes **href**—Specifies the location of the imported style sheet.

Usage Examples See all examples listed in "Commit Script Examples" on page 201.

Related Documentation

- **xsl:stylesheet on page 104**

xsl:otherwise

Syntax

```
<xsl:otherwise>
  ...
</xsl:otherwise>
```

Description Within an **<xsl:choose>** instruction, include the instructions that are processed if none of the expressions defined in the **test** attributes of the **<xsl:when>** elements evaluate as TRUE.

Usage Examples See "Example: Configuring Dual Routing Engines" on page 252 and "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

Related Documentation

- **xsl:choose on page 100**
- **xsl:when on page 107**

xsl:param

Syntax

```
<xsl:param name="qualified-name" select="expression">
  ...
</xsl:param>
```

Description Declare a parameter for a template (if it is within a template) or for the style sheet as a whole (if it is at the top level of the style sheet).

Attributes **name**—Defines the name of the parameter.

select—Defines the default value for the parameter, which is used if the person or client application that executes the script does not explicitly provide a value. The **select** attribute or the content of the **<xsl:param>** element can define the default value. Do not specify both a **select** attribute and some content; we recommend using the **select** attribute so as not to create a result tree fragment.

Usage Examples See "Example: Requiring and Restricting Configuration Statements" on page 201, "Example: Imposing a Minimum MTU Setting" on page 207, "Example: Limiting the Number of E1 Interfaces" on page 209, "Example: Limiting the Number of ATM Virtual Circuits" on page 217, and "Example: Preventing Import of the Full Routing Table" on page 256.

Related Documentation

- **xsl:template on page 105**
- **xsl:variable on page 107**
- **xsl:with-param on page 108**

xsl:stylesheet

Syntax

```
<xsl:stylesheet version="1.0">
  <xsl:import href="../import/junos.xsl"/>
  ...
</xsl:stylesheet>
```

Description Include the document element for the style sheet. Contains all the top-level elements such as global variable and parameter declarations, import elements, and templates. Any **<xsl:import>** elements must be the first elements within the style sheet, the first children of the **<xsl:stylesheet>** document element. The path can be any URI. The **../import/junos.xsl** path shown in the syntax is standard for all commit scripts, op scripts, and event scripts.

Attributes **version**—Specifies the version of XSLT that is being used. The Junos OS supports XSLT version 1.0.

Usage Examples See all examples listed in "Commit Script Examples" on page 201.

Related Documentation

- **xsl:import on page 103**

xsl:template

Syntax

```
<xsl:template match="pattern" mode="qualified-name" name="qualified-name">
  <xsl:param name="qualified-name" select="expression">
    ...
  </xsl:param>
  ...
</xsl:stylesheet>
```

Description Declare a template that contains rules to apply when a specified node is matched. The **match** attribute associates the template with an XML element. The **match** attribute can also be used to define a template for a whole branch of the XML document For example, **match="/"** matches the whole document.

When templates are applied to a node set using the **<xsl:apply-templates>** instruction, they might be applied in a particular mode; the **mode** attribute in the **<xsl:template>** instruction indicates the mode in which a template needs to be applied for the template to be used. If templates are applied in the specified mode, the **match** attribute is used to determine whether the template can be used with the particular node.

You can pass templates parameters by using the **<xsl:with-param>** element. To receive a parameter, the template must contain an **<xsl:param>** element that declares a parameter of that name. These parameters are listed before the body of the template, which is used to process the node and create a result.

Attributes **match**—Applies the template to nodes by specifying a pattern against which nodes are matched.

mode—Indicates the mode in which a template needs to be applied for the template to be used.

name—Calls the template by name.

Usage Examples See all examples listed in "Commit Script Examples" on page 201.

Related Documentation

- **xsl:apply-templates on page 99**
- **xsl:call-template on page 100**

xsl:text

Syntax

```
<xsl:text>
  ...
</xsl:text>
```

Description Insert literal text in the output.

Usage Examples See “Example: Requiring and Restricting Configuration Statements” on page 201, “Example: Imposing a Minimum MTU Setting” on page 207, “Example: Limiting the Number of E1 Interfaces” on page 209, “Example: Controlling IS-IS and MPLS Interfaces” on page 221, and “Example: Adding a Final then accept Term to a Firewall” on page 234.

xsl:value-of

Syntax <xsl:value-of select="*string-expression*"/>

Description Extract data from the XML structure. The **select** attribute specifies the expression that is evaluated. In the string expression, use @ to access attributes of elements. Use “ **.**” to access the contents of the element itself. If the result is a node set, the **<xsl:value-of>** instruction adds the string value of the first node in that node set; none of the structure of the node is preserved. To preserve the structure of the node, you must use the **<xsl:copy-of>** instruction instead.

Attributes **select**—Specifies the expression that is evaluated.

Usage Examples See “Example: Imposing a Minimum MTU Setting” on page 207, “Example: Limiting the Number of E1 Interfaces” on page 209, “Example: Controlling IS-IS and MPLS Interfaces” on page 221, “Example: Configuring Administrative Groups for LSPs” on page 248, and “Example: Automatically Configuring Logical Interfaces and IP Addresses” on page 259.

Related Documentation

- **xsl:copy-of on page 101**

xsl:variable

Syntax

```
<xsl:variable name="qualified-name" select="expression">
  ...
</xsl:variable>
```

Description Declare a local or global variable. If the **<xsl:variable>** instruction appears at the top level of the style sheet as a child of the **<xsl:stylesheet>** document element, it is a global variable with a scope covering the entire style sheet. Otherwise, it is a local variable with a scope of its following siblings and their descendants.

Attributes **name**—Specifies the name of the variable. After declaration, the variable can be referred to within XPath expressions using this name, prefixed with the **$** character.

select—Determines the value of the variable. The value of the variable is determined either by the **select** attribute or by the contents of the **<xsl:variable>** element. Do not specify both a **select** attribute and some content; we recommend using the **select** attribute so as not to create a result tree fragment.

Usage Examples See "Example: Limiting the Number of E1 Interfaces" on page 209, "Example: Limiting the Number of ATM Virtual Circuits" on page 217, "Example: Configuring Administrative Groups for LSPs" on page 248, and "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

Related Documentation

- **xsl:param on page 104**

xsl:when

Syntax

```
<xsl:when test="boolean-expression">
  ...
</xsl:when>
```

Description Within an **<xsl:choose>** instruction, specify a set of processing that occurs when the expression specified in the **test** attribute evaluates as TRUE. The XSLT processor processes only the instructions contained in the first **<xsl:when>** element whose **test** attribute evaluates as TRUE. If none of the **<xsl:when>** elements' **test** attributes evaluate as TRUE, the content of the **<xsl:otherwise>** element, if there is one, is processed.

Attributes **test**—Specifies a Boolean expression.

Usage Examples See "Example: Configuring Dual Routing Engines" on page 252, "Example: Preventing Import of the Full Routing Table" on page 256, and "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

Related Documentation

- **xsl:choose on page 100**
- **xsl:if on page 102**
- **xsl:otherwise on page 103**

xsl:with-param

Syntax

```
<xsl:with-param name="qualified-name" select="expression">
  ...
</xsl:with-param>
```

Description Specify the value of a parameter to be passed into a template. It can be used when applying templates with the **<xsl:apply-templates>** instruction or calling templates with the **<xsl:call-template>** instruction.

Attributes **name**—Specifies the name of the parameter for which the value is being passed.

select—Determines the value of the parameter. The value of the parameter is determined either by the **select** attribute or by the contents of the **<xsl:with-param>** element. Do not specify both a **select** attribute and some content. We recommend using the **select** attribute to set the parameter so as to prevent the parameter from being passed a result tree fragment as its value.

Usage Examples See "Example: Configuring Dual Routing Engines" on page 252, "Example: Preventing Import of the Full Routing Table" on page 256, and "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

Related Documentation

- **xsl:apply-templates on page 99**
- **xsl:call-template on page 100**
- **xsl:param on page 104**

CHAPTER 8

Summary of SLAX Statements

This chapter summarizes the Stylesheet Language Alternative Syntax (SLAX) statements, with brief examples and Extensible Stylesheet Language Transformations (XSLT) equivalents. The statements are organized alphabetically.

apply-templates

Syntax apply-templates *expression*;

Description Apply one or more templates, according to the value of the node-set expression. The templates that are applied are passed the parameters specified by the **with** statement within the **apply-templates** statement.

Attributes *expression*—Selects the nodes to which the processor applies templates. By default, the processor applies templates to the child nodes of the current node.

SLAX Example

```
match configuration {
  apply-templates system/host-name;
}
```

XSLT Equivalent

```
<xsl:template match="configuration">
  <xsl:apply-templates select="system/host-name"/>
</xsl:template>
```

Usage Examples See "Example: Adding a Final then accept Term to a Firewall" on page 234 and "Example: Preventing Import of the Full Routing Table" on page 256.

Related Documentation

- **match on page 115**
- **mode on page 116**
- **with on page 122**

call

Syntax

```
call template-name (parameter-name = value) {
  /* code */
}
```

Description

Invoke a template. You can include a comma-separated list of parameters, with the parameter name and an optional equal sign (=) and value expression. If the value is not given, the current value of the parameter is passed.

You can declare additional parameters inside the code block using the **with** statement.

SLAX Example

```
match configuration {
  var $name-servers = name-servers/name;
  call temp();
  call temp($name-servers, $size = count($name-servers));
  call temp() {
    with $name-servers;
    with $size = count($name-servers);
  }

  template temp($name-servers, $size = 0) {
    <output> "template called with size " _ $size;
  }
}
```

XSLT Equivalent

```
<xsl:template match="configuration">
  <xsl:variable name="name-servers" select="name-servers/name"/>
  <xsl:call-template name="temp"/>
  <xsl:call-template name="temp">
    <xsl:with-param name="name-servers" select="$name-servers"/>
    <xsl:with-param name="size" select="count($name-servers)"/>
  </xsl:call-template>
  <xsl:call-template name="temp">
    <xsl:with-param name="name-servers" select="$name-servers"/>
    <xsl:with-param name="size" select="count($name-servers)"/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="temp">
  <xsl:param name="name-servers"/>
  <xsl:param name="size" select="0"/>
  <output>
    <xsl:value-of select="concat('template called with size ', $size)"/>
  </output>
</xsl:template>
```

Usage Examples

See “Example: Requiring and Restricting Configuration Statements” on page 201, “Example: Imposing a Minimum MTU Setting” on page 207, and “Example: Automatically Configuring Logical Interfaces and IP Addresses” on page 259.

Related Documentation
- template on page 119
- with on page 122

else

Syntax

```
else {
  /* code */
}

else {
  if (expression) {
    /* code */
  }
}
```

Description

Include the instructions that are processed if none of the expressions defined in the **test** attributes of the **if** statement evaluate as TRUE.

SLAX Example

```
if (starts-with(name, "fe-")) {
  if (mtu < 1500) {
    /* Select the Fast Ethernet interfaces with low MTUs */
  }
}
else {
  if (mtu > 8096) {
    /* Select the non-Fast Ethernet interfaces with high MTUs */
  }
}
```

XSLT Equivalent

```
<xsl:choose>
  <xsl:when select="starts-with(name, 'fe-')">
    <xsl:if test="mtu &lt; 1500">
      <!-- Select with Fast Ethernet interfaces with low MTUs -->
    </xsl:if>
  </xsl:when>
  <xsl:otherwise>
    <xsl:if test="mtu &gt; 8096">
      <!-- Select the non-Fast Ethernet interfaces with high MTUs -->
    </xsl:if>
  </xsl:otherwise>
</xsl:choose>
```

Usage Examples

See "Example: Configuring Dual Routing Engines" on page 252 and "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

Related Documentation
- if on page 114

for-each

Syntax

```
for-each (expression) {
  /* code */
}
```

Description

Include a looping mechanism that repeats script processing for each instance of identical XML elements. The element nodes are selected by the ***expression*** attribute. The instructions contained in the **for-each** statement processes each of the nodes.

The statement consists of the **for-each** keyword, the parentheses-delimited expression, and a curly braces-delimited block. The SLAX **for-each** statement functions like the **<xsl:for-each>** element.

Attributes

expression—Selects the nodes to which the processor applies templates. By default, the processor applies templates to the child nodes of the current node.

SLAX Example

```
for-each ($inventory/chassis/chassis-module/
      chassis-sub-module[part-number == '750-000610']) {
  <message> "Down rev PIC in " _ ../name _ ", " _ name _ ": " _ description;
}
```

XSLT Equivalent

```
<xsl:for-each select="$inventory/chassis/chassis-module/
      chassis-sub-module[part-number == '750-000610']">
  <message>
    <xsl:value-of select="concat('Down rev PIC in ', ../name, ', ', name, ': ',
        description)"/>
  </message>
</xsl:for-each>
```

Usage Examples

See “Example: Requiring and Restricting Configuration Statements” on page 201, “Example: Imposing a Minimum MTU Setting” on page 207, “Example: Limiting the Number of E1 Interfaces” on page 209, “Example: Adding T1 Interfaces to a RIP Group” on page 224, “Example: Configuring Administrative Groups for LSPs” on page 248, and “Example: Configuring Dual Routing Engines” on page 252.

if

Syntax

```
if (expression) {
  /* code */
}
```

Description Include a conditional construct that causes instructions to be processed if the Boolean expression held in the **test** attribute evaluates to TRUE.

Attributes *expression*—Selects the nodes to which the processor applies templates. By default, the processor applies templates to the child nodes of the current node.

SLAX Example

```
if (starts-with(name, "fe-")) {
  if (mtu < 1500) {
    /* Select the Fast Ethernet interfaces with low MTUs */
  }
}
else {
  if (mtu > 8096) {
    /* Select the non-Fast Ethernet interfaces with high MTUs */
  }
}
```

XSLT Equivalent

```
<xsl:choose>
  <xsl:when select="starts-with(name, 'fe-')">
    <xsl:if test="mtu &lt; 1500">
      <!-- Select with Fast Ethernet interfaces with low MTUs -->
    </xsl:if>
  </xsl:when>
  <xsl:otherwise>
    <xsl:if test="mtu &gt; 8096">
      <!-- Select the non-Fast Ethernet interfaces with high MTUs -->
    </xsl:if>
  </xsl:otherwise>
</xsl:choose>
```

Usage Examples See “Example: Configuring Dual Routing Engines” on page 252, “Example: Preventing Import of the Full Routing Table” on page 256, and “Example: Automatically Configuring Logical Interfaces and IP Addresses” on page 259.

Related Documentation

- **else on page 112**

match

Syntax

```
match expression {
statements;
}
```

Description Declare a template that contains rules to apply when a specified node is matched. The **match** statement associates the template with an XML element. The **match** statement can also be used to define a template for a whole branch of the XML document For example, **match /** matches the whole document.

Attributes *expression*—Specifies a pattern against which nodes are matched.

SLAX Example

```
match host-name {
  <hello> .;
}
```

XSLT Equivalent

```
<xsl:template match="host-name">
  <hello>
    <xsl:value-of select="."/>
  </hello>
</xsl:template>
```

Usage Examples See all examples listed in “Commit Script Examples” on page 201.

Related Documentation

- **apply-templates on page 110**
- **mode on page 116**

mode

Syntax mode *qualified-name*;

Description Indicate the mode in which a template needs to be applied for the template to be used. If templates are applied in the specified mode, the **match** statement is used to determine whether the template can be used with the particular node.

This statement is comparable to the **mode** attribute of the **<xsl:template>** element. You can include this statement inside a SLAX **match** or **apply-templates** statement.

SLAX Example

```
match * {
  mode "one";
  <one> .;
}

match * {
  mode "two";
  <two> string-length(.);
}

match / {
  apply-templates version {
    mode "one";
  }
  apply-templates version {
    mode "two";
  }
}
```

XSLT Equivalent

```
<xsl:template match="*" mode="one">
  <one>
    <xsl:value-of select="."/>
  </one>
</xsl:template>

<xsl:template match="*" mode="two">
  <two>
    <xsl:value-of select="string-length(.)"/>
  </two>
</xsl:template>

<xsl:template match="/">
  <xsl:apply-templates select="version" mode="one"/>
  <xsl:apply-templates select="version" mode="two"/>
</xsl:template>
```

Usage Examples See “Example: Adding a Final then accept Term to a Firewall” on page 234.

Related Documentation

- **apply-templates on page 110**
- **match on page 115**

param

Syntax param $*name*=*value*;

Description Declare a parameter for a template (within a template) or for the script as a whole (at the top level of the script). You can include an initial value by following the variable name with an equal sign (=) and a value expression.

In SLAX, parameter and variable names contain the dollar sign ($) even in the declaration. This is unlike the **name** attribute of **<xsl:variable>** and **<xsl:parameter>** elements.

Attributes $*name*—Defines the name of the parameter.

value—Defines the default value for the parameter, which is used if the person or client application that executes the script does not explicitly provide a value.

SLAX Example

```
param $vrf;
param $dot = .;
```

XSLT Equivalent

```
<xsl:param name="vrf"/>
<xsl:param name="dot" select="."/>
```

Usage Examples See "Example: Requiring and Restricting Configuration Statements" on page 201, "Example: Imposing a Minimum MTU Setting" on page 207, "Example: Limiting the Number of E1 Interfaces" on page 209, "Example: Limiting the Number of ATM Virtual Circuits" on page 217, and "Example: Preventing Import of the Full Routing Table" on page 256.

Related Documentation

- **var on page 120**

priority

Syntax priority *number*;

Description If more than one template matches a node in the specified mode, this statement determines which template is used. The highest priority wins. If no priority is specified explicitly, the priority of a template is determined by the **match** statement.

This statement is comparable to the **priority** attribute of the **<xsl:template>** element. You can include this statement inside a SLAX **match** statement.

SLAX Example

```
match * {
  priority 10;
  <output> .;
}
```

XSLT Equivalent

```
<xsl:template match="*" priority="10">
  <output>
    <xsl:value-of select="."/>
  </output>
</xsl:template>
```

Usage Examples None of the examples in this manual use this statement.

Related Documentation

- **apply-templates on page 110**
- **match on page 115**

template

Syntax

```
template qualified-name (parameter-name = value) {
  /* code */
}
```

Description

Declare a template. You can include a comma-separated list of parameter declarations, with the parameter name and an optional equal sign (=) and value expression. You can declare additional parameters inside the code block using the **param** statement. You can invoke the template using the **call** statement.

SLAX Example

```
match configuration {
  var $name-servers = name-servers/name;
  call temp();
  call temp($name-servers, $size = count($name-servers));
  call temp() {
    with $name-servers;
    with $size = count($name-servers);
  }

  template temp($name-servers, $size = 0) {
    <output> "template called with size " _ $size;
  }
}
```

XSLT Equivalent

```
<xsl:template match="configuration">
  <xsl:variable name="name-servers" select="name-servers/name"/>
  <xsl:call-template name="temp"/>
  <xsl:call-template name="temp">
    <xsl:with-param name="name-servers" select="$name-servers"/>
    <xsl:with-param name="size" select="count($name-servers)"/>
  </xsl:call-template>
  <xsl:call-template name="temp">
    <xsl:with-param name="name-servers" select="$name-servers"/>
    <xsl:with-param name="size" select="count($name-servers)"/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="temp">
  <xsl:param name="name-servers"/>
  <xsl:param name="size" select="0"/>
  <output>
    <xsl:value-of select="concat('template called with size ', $size)"/>
  </output>
</xsl:template>
```

Usage Examples

See all examples listed in “Commit Script Examples” on page 201.

Related Documentation

- call on page 111
- with on page 122

var

Syntax var $*name*=*value*;

Description Declare a local or global variable. If the **var** statement appears at the top of the script, it is a global variable with a scope covering the entire script. Otherwise, it is a local variable. You can include an initial value by following the variable name with an equal sign (**=**) and a value expression.

In SLAX, parameter and variable names contain the dollar sign (**$**) even in the declaration. This is unlike the **name** attribute of **<xsl:variable>** and **<xsl:parameter>** elements.

Attributes $*name*—Specifies the name of the variable. After declaration, the variable can be referred to within expressions using this name, including the **$** character.

value—Defines the default value for the variable, which is used if the person or client application that executes the script does not explicitly provide a value.

SLAX Example

```
var $vrf;
var $location = $dot/@location;
var $message = "We are in "_ $location _" now.";
```

XSLT Equivalent

```
<xsl:variable name="vrf"/>
<xsl:variable name="location" select="$dot/location"/>
<xsl:variable name="message" select="concat('We are in ', $location, now.')"/>
```

Usage Examples See “Example: Limiting the Number of E1 Interfaces” on page 209, “Example: Limiting the Number of ATM Virtual Circuits” on page 217, “Example: Configuring Administrative Groups for LSPs” on page 248, and “Example: Automatically Configuring Logical Interfaces and IP Addresses” on page 259.

Related Documentation

- **param on page 117**

version

Syntax version 1.0;

Description Specify the version of SLAX that is being used. All SLAX style sheets must begin with a **version** statement.

Version 1.0 uses XML version 1.0 and XSLT version 1.1.

In addition, the **xsl** namespace is implicitly defined as follows:

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

Attributes ***version-number***—Specifies the version of SLAX. The Junos OS supports SLAX version 1.0.

SLAX Example

```
version 1.0;
```

XSLT Equivalent

```
<xsl:stylesheet version="1.0">
```

Usage Examples See all examples listed in “Commit Script Examples” on page 201.

with

Syntax with *name* = *value*;

Description Specify a variable or parameter to be passed into a template. You can use this statement when you apply templates with the **apply-templates** statement or call templates with the **match** statement.

Optionally, you can specify a value for the parameter by including an equal sign (=) and a value expression. If no value is given, the current value of the variable or parameter is passed.

Attributes ***name***—Name of the variable or parameter for which the value is being passed.

value—Value of the parameter being passed to the template.

SLAX Example

```
match configuration {
  var $domain = domain-name;
  apply-templates system/host-name {
    with $message = "Invalid host-name";
    with $domain;
  }
}

match host-name {
  param $message = "Error";
  param $domain;
  <hello> $message _ ":: " _ . _ " (" _ $domain _ ")";
}
```

XSLT Equivalent

```
<xsl:template match="configuration">
  <xsl:apply-templates select="system/host-name">
    <xsl:with-param name="message" select="'Invalid host-name'"/>
    <xsl:with-param name="domain" select="$domain"/>
  </xsl:apply-templates>
</xsl:template>

<xsl:template match="host-name">
  <xsl:param name="message" select="'Error'"/>
  <xsl:param name="domain"/>
  <hello>
    <xsl:value-of select="concat($message, ':: ', ., ' (', $domain, ')')"/>
  </hello>
</xsl:template>
```

Usage Examples See “Example: Configuring Dual Routing Engines” on page 252, “Example: Preventing Import of the Full Routing Table” on page 256, and “Example: Automatically Configuring Logical Interfaces and IP Addresses” on page 259.

Related Documentation

- **apply-templates on page 110**
- **match on page 115**

PART 2

Configuration Automation

CHAPTER 9

Commit Scripts Overview

This chapter includes the following topics:

Commit Script Overview

Junos OS commit scripts enforce custom configuration rules during the commit process. When a candidate configuration is committed, it is inspected by each active commit script. If a configuration violates your custom rules, the script can instruct the Junos OS to take appropriate action. A commit script can perform the following actions:

- Generate and display custom warning messages to the user
- Generate and log custom system log (syslog) messages
- Change the configuration to conform to the custom business rules
- Generate a commit error and halt the commit operation

Commit scripts are based on the Junos XML management protocol and the Junos XML API. The Junos XML management protocol is an XML based RPC mechanism, and the Junos XML API is an XML representation of Junos configuration statements and operational mode commands. Commit scripts can be written in either the Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) scripting language. Commit scripts use the XML Path Language (XPath) to locate the configuration objects to be inspected and XSLT or SLAX constructs to specify the actions to perform on the configuration objects. The actions can change the configuration or generate messages about it. For more information about XSLT, see “XSLT Overview” on page 19. For more information about SLAX, see “SLAX Overview” on page 35

Additionally, you can create *macros*, which allow you to create custom configuration syntax that simplifies the task of configuring a device running Junos OS. By itself, your custom syntax has no operational impact on the device. A corresponding commit script

macro uses your custom syntax as input data for generating standard Junos configuration statements that execute your intended operational impact.

To view the device's current configuration in the Extensible Markup Language (XML), using the command-line interface's (CLI's) operational mode, issue the **show configuration | display xml** command. To view your configuration in commit-script-style XML, issue the **show configuration | display commit-scripts view** command.

Related Documentation

- Junos XML API and Junos XML Management Protocol Overview on page 15
- SLAX Overview on page 35
- XSLT Overview on page 19

Advantages of Using Commit Scripts

Reducing human error in a network configuration can significantly improve network uptime. Commit scripts enable you to control operational practices and enforce operational policy, thereby decreasing the possibility of human error. Restricting device configurations in accordance with custom design rules can vastly improve network reliability.

Consider the following examples of actions you can perform with commit scripts:

- Basic sanity test—Ensure that the **[edit interfaces]** and **[edit protocols]** hierarchies have not been accidentally deleted.
- Consistency check—Ensure that every T1 interface configured at the **[edit interfaces]** hierarchy level is also configured at the **[edit protocols rip]** hierarchy level.
- Dual Routing Engine configuration test—Ensure that the **re0** and **re1** configuration groups are set up correctly. When you use configuration groups, the inherited values can be overridden in the target configuration. A commit script can determine if an individual target configuration element is blocking proper inheritance of the configuration group settings.
- Interface density—Ensure that a channelized interface does not have too many channels configured.
- Link scaling—Ensure that SONET/SDH interfaces never have a maximum transmission unit (MTU) size less than 4 kilobytes (KB).
- Import policy check—Ensure that an interior gateway protocol (IGP) does not use an import policy that imports the full routing table.
- Cross-protocol checks—Ensure that all LDP-enabled interfaces are configured for an IGP, or ensure that all IGP-enabled interfaces are configured for LDP.
- IGP design check—Ensure that Level 1 IS-IS routers are never enabled.

When a candidate configuration does not adhere to your design rules, a commit script can instruct the Junos OS to generate custom warnings, system log messages, or error messages that block the commit operation from succeeding. In addition, the commit

script can change the configuration in accordance with your rules and then proceed with the commit operation.

Consider a network design that requires every interface on which the International Organization for Standardization (ISO) family of protocols is enabled to also have MPLS enabled. At commit time, a commit script inspects the configuration and issues an error if this requirement is not met. This error causes the commit operation to fail and forces the user to update the configuration to comply.

Instead of an error, the commit script can issue a warning about the configuration problem and then automatically correct it by changing the configuration to enable MPLS on all interfaces. A system log message can also be generated, indicating that corrective action was taken.

Another option is to define a macro that enables ISO protocols and MPLS when the macro is applied to an interface. Configuring this macro simplifies the configuration task while ensuring that both protocols are configured together.

Finally, you can have the commit script correct the configuration using a *transient change*. In our example, a transient change allows MPLS to always be enabled on ISO-enabled interfaces without having the configuration statements appear in the candidate configuration.

All of these example scenarios are included in “Commit Script Examples” on page 201.

NOTE: Transient changes cause a change to be generated in the *checkout configuration* but not in the candidate configuration. The checkout configuration is the configuration database that is checked for standard Junos OS syntax just before a configuration becomes active. This means transient changes are not saved in the configuration if the associated commit script is deleted or deactivated. The **show configuration | display commit-scripts** command displays all the statements that are in the configuration, including statements that were generated by transient changes. For more information, see “Overview of Generating Persistent or Transient Configuration Changes” on page 155.

How Commit Scripts Work

You enable commit scripts by listing the names of one or more commit script files at the **[edit system scripts commit]** hierarchy level. These scripts contain instructions that enforce custom configuration rules. Commit scripts are invoked during the commit process before the standard Junos OS validity checks are performed.

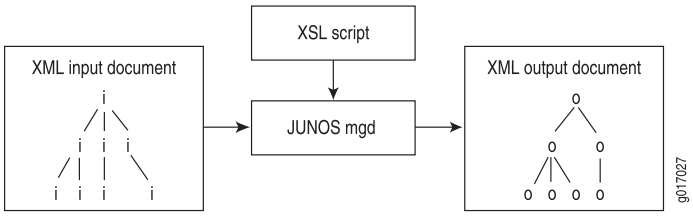
When you perform a commit operation, the Junos OS executes each script in turn, passing the information in the candidate configuration to the scripts. The script inspects the configuration, performs the necessary tests and validations, and generates a set of instructions for performing certain actions. These actions include generating error, warning, and system log messages. If errors are generated, the commit operation fails and the

candidate configuration remains unchanged. This is the same behavior that occurs with standard commit errors.

Commit scripts can also generate changes to the system configuration. Because the changes are loaded before the standard validation checks are performed, they are validated for correct syntax, just like statements already present in the configuration before the script is applied. If the syntax is correct, the configuration is activated and becomes the active, operational device configuration.

Figure 4 on page 128 shows the flow of commit script input and output.

Figure 4: Commit Script Input and Output



The following sections discuss several important concepts related to the commit script input and output:

- Commit Script Input on page 128
- Commit Script Output on page 129
- Commit Scripts and the Junos OS Commit Model on page 130
- Avoiding Potential Conflicts When Using Multiple Commit Scripts on page 132

Commit Script Input

The input for a commit script is the postinheritance candidate configuration in Junos XML API format. The term *postinheritance* means that all configuration group values have been inherited by their targets in the candidate configuration and the inactive portions of the configuration have been removed. For more information about configuration groups, see the *Junos OS CLI User Guide*.

When you issue the **commit** command, the Junos OS automatically generates the candidate configuration in XML format and reads it into the management (mgd) process, at which time the input is evaluated by any commit scripts.

To display the XML format of the postinheritance configuration, issue the **show | display commit-scripts view** command:

```
[edit]
user@host# show | display commit-scripts view
```

To display all configuration groups data, including script-generated changes to the groups, issue the **show groups | display commit-scripts** command:

```
[edit]
user@host# show groups | display commit-scripts
```

To save the commit script input to a file, add the **save** command to the command line:

```
[edit]
user@host# show | display commit-scripts view | save filename.xml
```

By default, the file is placed in your home directory on the switch, router, or security device.

Commit Script Output

To specify the desired commit script output—including warning, error, and system log messages, persistent changes, and transient changes—the script can contain tags that appear in any order, in any number. The tags for specifying output are as follows:

- **<xnm:warning>**—Generates a warning message
- **<xnm:error>**—Generates an error message.
- **<syslog><message>**—Generates a system log message.
- **<change>**—Generates a persistent change to the configuration.
- **<transient-change>**—Generates a transient change to the configuration.
- **<xsl:call-template name="jcs:emit-change">**
  <xsl:with-param name="content">—Generates a persistent change relative to the current context node as defined by an XPath expression.
- **<xsl:call-template name="jcs:emit-change">**
  <xsl:with-param name="tag" select="'transient-change'"/>
  <xsl:with-param name="content">—Generates a transient change relative to the current context node as defined by an XPath expression.
- **<xsl:call-template name="jcs:emit-change">**
  <xsl:with-param name="message">
  <xsl:text>—Generates a warning message in conjunction with a configuration change. You can use this set of tags to generate a notification that the configuration has been changed.

The Junos OS processes this output and performs the appropriate actions. Errors and warnings are passed back to the Junos OS CLI or to a Junos XML protocol client application. The presence of an error automatically causes the commit operation to fail. Persistent and transient changes are loaded into the appropriate configuration database.

To test the output of error, warning, and system log messages from commit scripts, issue the **commit check | display xml** command:

```
[edit]
user@host# commit check | display xml
```

To display a detailed trace of commit script processing, issue the **commit check | display detail** command:

```
[edit]
user@host# commit check | display detail
```

NOTE: System log messages do not appear in the trace output, so you cannot use the commit check operation to test script-generated system log messages. Furthermore, system log messages are written to the system log during a commit operation, but not during a commit check operation.

Related Documentation

- **jcs:emit-change Template on page 83**

Commit Scripts and the Junos OS Commit Model

The Junos OS uses a commit model to update the device's configuration. This model allows you to make a series of changes to a candidate configuration without affecting the operation of the device. When the changes are complete, you can commit the configuration. The commit operation saves the candidate configuration changes into the current configuration.

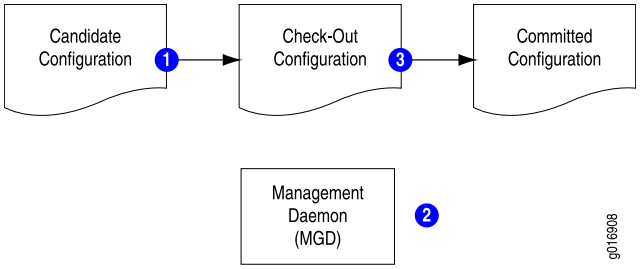
When you commit a set of changes in the candidate configuration, two methods are used to forward these changes to the current configuration:

- Standard commit model—Used when no commit scripts are active on the device.
- Commit script model—Incorporates commit scripts into the commit model.

Standard Commit Model

In the standard commit model, the management (mgd) process validates the candidate configuration based on standard Junos validation rules. If the configuration file is valid, it becomes the current active configuration. Figure 5 on page 130 and the accompanying discussion explain how the standard commit model works:

Figure 5: Standard Commit Model



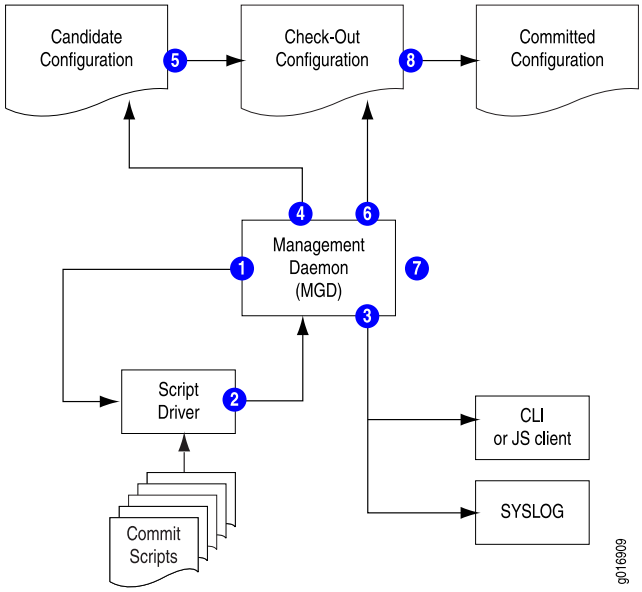
In the standard commit model, the software performs the following steps:

1. When the candidate configuration is committed, it is copied to become the checkout configuration.
2. The mgd process validates the checkout configuration.
3. If no error occurs, the checkout configuration is copied as the current active configuration.

Commit Model with Commit Scripts

When commit scripts are added to the standard commit model, the process becomes more complex. The mgd process first passes an XML-formatted checkout configuration to a script driver, which handles the verification of the checkout configuration by the commit scripts. When verification is complete, the script driver returns an XML *action file* to the mgd process. The mgd process follows the instructions in the action file to update the candidate and checkout configurations, issue messages to the CLI, and write information to the system log as required. After processing the action file, the mgd process performs the standard Junos OS validation. Figure 6 on page 131 and the accompanying discussion explain this process.

Figure 6: Commit Model with Commit Scripts Added



In the commit script model, the Junos OS performs the following steps:

1. When the candidate configuration is committed, the mgd process sends the XML-formatted candidate configuration to the script driver.
2. Each enabled commit script is invoked against the candidate configuration, and each script can generate a set of actions for the mgd process to perform. The action are collected in an XML action file.
3. The mgd process performs the following actions in response to **<error>**, **<warning>**, and **<syslog>** tag elements in the action file:
   - **<error>**—The mgd process halts the commit process (that is, the commit operation fails), returns an error message to the CLI or Junos XML protocol client, and takes no further action.
   - **<warning>**—The mgd process forwards the message to the CLI or the Junos XML protocol client.
   - **<syslog>**—The mgd process forwards the message to the system log process.

4. If the action file includes any **<change>** tag elements, the mgd process loads the requested changes into the candidate configuration.
5. The candidate configuration is copied to become the checkout configuration.
6. If the action file includes any **<transient-change>** tag elements, the mgd process loads the requested changes into the checkout configuration.
7. The mgd process validates the checkout configuration.
8. If there are no validation errors, the checkout configuration is copied to become the current active configuration.

Changes that are made to the candidate configuration during the commit operation are not evaluated by the custom rules during that commit operation. However, persistent changes are maintained in the candidate configuration and are evaluated by the custom rules during subsequent commit operations. For more information about how commit scripts change the candidate configuration, see "Avoiding Potential Conflicts When Using Multiple Commit Scripts" on page 132.

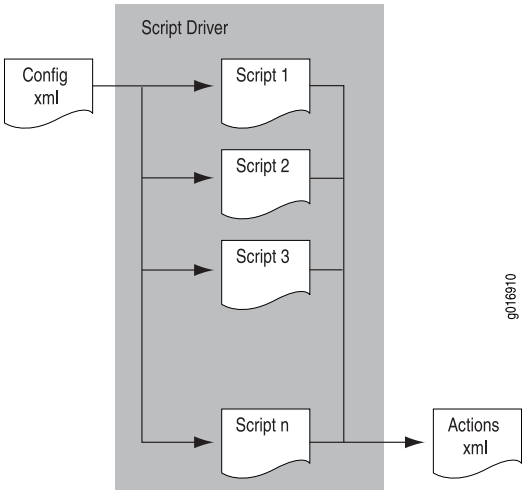
Transient changes are never evaluated by the custom rules in commit scripts, because they are made to the checkout configuration only after the commit scripts have evaluated the candidate configuration and the candidate is copied to become the checkout configuration. To remove a transient change from the configuration, remove, disable, or deactivate the commit script (as discussed in "Controlling Execution of Commit Scripts During Commit Operations" on page 186), or comment out the code that generates the transient change.

For more information about differences between persistent and transient changes, see "Overview of Generating Persistent or Transient Configuration Changes" on page 155.

Avoiding Potential Conflicts When Using Multiple Commit Scripts

When you use multiple commit scripts, each script evaluates the original candidate configuration file. Changes made by one script are not evaluated by the other scripts. This means that conflicts between scripts might not be resolved when the scripts are first applied to the configuration. The commit scripts are executed in the order they are listed at the **[edit system scripts commit]** hierarchy level, as illustrated in Figure 7 on page 133.

Figure 7: Configuration Evaluation by Multiple Commit Scripts



As an example of a conflict between commit scripts, suppose that commit script **A.xsl** is created to ensure that the device uses the domain name (DNS) server with IP address **192.168.0.255**. Later, the DNS server's address is changed to **192.168.255.255** and a second script, **B.xsl**, is added to check that the device uses the DNS server with that address. However, script **A.xsl** is not removed or disabled.

Because each commit script evaluates the original candidate configuration, the final result of executing both scripts **A.xsl** and **B.xsl** depends on which DNS server address is configured in the original candidate configuration. If the now outdated address of **192.168.0.255** is configured, script **B.xsl** changes it to **192.168.255.255**. However, if the correct address of **192.168.255.255** is configured, script **A.xsl** changes it to the incorrect value **192.168.0.255**.

Exercise care to ensure that you do not introduce conflicts between scripts like those described in the example. As a method of checking for conflicts with persistent changes, you can issue two separate **commit** commands.

Required Boilerplate for Commit Scripts

When you write commit scripts, you use Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) tools provided with the Junos OS. These tools include basic boilerplate that you must include in all commit scripts, optional extension functions that accomplish scripting tasks more easily, and named templates that make commit scripts easier to read and write, which you import from a file called junos.xsl. For more information about the extension functions and templates, see "Junos Extension Functions in the jcs Namespace" on page 62.

Commit scripts are based on Junos XML and Junos XML protocol tag elements. Like all XML elements, angle brackets enclose the name of a Junos XML or Junos XML protocol tag element in its opening and closing tags. This is an XML convention, and the brackets are a required part of the complete tag element name. They are not to be confused with the angle brackets used in the documentation to indicate optional parts of Junos OS CLI command strings.

You must include either XSLT or SLAX boilerplate as the starting point for all commit scripts that you create. The XSLT boilerplate follows:

XSLT Boilerplate for Commit Scripts

```
1  <?xml version="1.0" standalone="yes"?>
2  <xsl:stylesheet version="1.0"
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4     xmlns:junos="http://xml.juniper.net/junos/*/junos"
5     xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
6     xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
7     <xsl:import href="../import/junos.xsl"/>

8     <xsl:template match="configuration">
        <!-- ... Insert your code here ... -->
9     </xsl:template>
10 </xsl:stylesheet>
```

Line 1 is the Extensible Markup Language (XML) processing instruction (PI). This PI specifies that the code is written in XML using version 1.0. The XML PI, if present, must be the first noncomment token in the script file.

```
1  <?xml version="1.0"?>
```

Lines 2 through 6 set the style sheet element and the associated namespaces. Line 2 sets the style sheet version as 1.0. Lines 3 through 6 list all the namespace mappings commonly used in commit scripts. Not all of these prefixes are used in this example, but it is not an error to list namespace mappings that are not referenced. Listing all namespace mappings prevents errors if the mappings are used in later versions of the script.

```
2  <xsl:stylesheet version="1.0"
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4     xmlns:junos="http://xml.juniper.net/junos/*/junos"
5     xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
6     xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
```

Line 7 is an XSLT import statement. It loads the templates and variables from the file referenced as **../import/junos.xsl**, which ships as part of the Junos OS. The junos.xsl file contains a set of named templates you can call in your scripts. These named templates are discussed in "Junos Named Templates in the jcs Namespace" on page 82.

```
7     <xsl:import href="../import/junos.xsl"/>
```

Line 8 defines a template that matches the **<configuration>** element, which is the node selected by the **<xsl:template match="/">** template, contained in the junos.xsl import file. The **<xsl:template match="configuration">** element allows you to exclude the **/configuration/** root element from all XML Path Language (XPath) expressions in the script and begin XPath expressions with the top Junos OS hierarchy level. For more information, see "XPath Overview" on page 21.

```
8     <xsl:template match="configuration">
```

Add your code between Lines 8 and 9.

Line 9 closes the template.

```
9     </xsl:template>
```

Line 10 closes the style sheet and the commit script.

```
10 </xsl:stylesheet>
```

SLAX Boilerplate for Commit Scripts

The corresponding SLAX boilerplate is as follows:

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  /*
  * Insert your code here
  */
}
```

Design Considerations for Commit Scripts

After you have an understanding of XSLT and some experience looking at Junos OS configuration data in XML, creating commit scripts is fairly straightforward. This section provides some advice and common patterns for developing commit scripts.

XSLT is an interpreted language, making performance an important consideration. For best performance, minimize node traversals and testing performed on each node. When possible, use the **select** attribute on a recursive **<xsl:apply-templates>** invocation to limit the portion of the document hierarchy being visited.

For example, the following **select** attribute limits the nodes to be evaluated by specifying SONET/SDH interfaces that have the **inet** (IPv4) protocol family enabled:

```
<xsl:apply-templates select="interfaces/interface[starts-with(name, 'so-') and
unit/family/inet]"/>
```

The following example contains two **<xsl:apply-templates>** instructions that limit the scope of the script to the **import** statements configured at the **[edit protocols ospf]** and **[edit protocols isis]** hierarchy levels:

```
<xsl:template match="configuration">
  <xsl:apply-templates select="protocols/ospf/import"/>
  <xsl:apply-templates select="protocols/isis/import"/>
  <!-- ... body of template ... -->
</xsl:template>
```

In an interpreted language, doing anything more than once can affect performance. If the script needs to reference a node or node set repeatedly, make a variable that holds the node set, and then make multiple references to the variable. For example, the following variable declaration creates a variable called **mpls** that resolves to the **[edit protocols mpls]** hierarchy level. This allows the script to traverse the **/protocols/** hierarchy searching for the **mpls/** node only once.

```
<xsl:variable name="mpls" select="/protocols/mpls"/>
<xsl:choose>
  <xsl:when test="$mpls/path-mtu/allow-fragmentation">
    <!-- ... -->
  </xsl:when>
  <xsl:when test="$mpls/hop-limit &gt; 40">
    <!-- ... -->
  </xsl:when>
```

```
</xsl:choose>
```

Variables are also important when using **<xsl:for-each>** instructions, because the current context node examines each node selected by the **<xsl:for-each>** instruction. For example, the following script uses multiple variables to store and refer to values as the **<xsl:for-each>** instruction evaluates the E1 interfaces that are configured on all channelized STM1 (**cstm1-**) interfaces:

```
<xsl:param name="limit" select="16"/>
<xsl:template match="configuration">
  <xsl:variable name="interfaces" select="interfaces"/>
  <xsl:for-each select="$interfaces/interface[starts-with(name, 'cstm1-')]">
    <xsl:variable name="triple" select="substring-after(name, 'cstm1-')"/>
    <xsl:variable name="e1name" select="concat('e1-', $triple)"/>
    <xsl:variable name="count"
          select="count($interfaces/interface[starts-with(name, $e1name)])"/>
    <xsl:if test="$count > $limit">
      <xnm:error>
        <edit-path>[edit interfaces]</edit-path>
        <statement><xsl:value-of select="name"/></statement>
        <message>
          <xsl:text>E1 interface limit exceeded on CSTM1 IQ PIC. </xsl:text>
          <xsl:value-of select="$count"/>
          <xsl:text> E1 interfaces are configured, but only </xsl:text>
          <xsl:value-of select="$limit"/>
          <xsl:text> are allowed.</xsl:text>
        </message>
      </xnm:error>
    </xsl:if>
  </xsl:for-each>
</xsl:template>
```

If you channelize a **cstm1-0/1/0** interface into 17 **E1** interfaces, the script causes the following error message to appear when you issue the **commit** command. (For more information about this example, see “Example: Limiting the Number of E1 Interfaces” on page 209.)

```
[edit]

user@host# commit
[edit interfaces]
    'cstm1-0/1/0'
    E1 interface limit exceeded on CSTM1 IQ PIC.
    17 E1 interfaces are configured, but only 16 are allowed.
error: 1 error reported by commit scripts
error: commit script failure
```

Line-by-Line Explanation of Sample Commit Scripts

The following examples illustrate how to construct commit scripts. Each example is followed by a line-by-line explanation.

Applying a Change to SONET/SDH Interfaces

The following commit script applies a transient change to each interface whose name begins with **so-**, setting the encapsulation to **ppp**. For information about transient changes, see "Overview of Generating Persistent or Transient Configuration Changes" on page 155. For a SLAX version of this example, see "Example: Generating a Transient Change" on page 168.

```
 1  <?xml version="1.0"?>
 2  <xsl:stylesheet version="1.0"
 3    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 4    xmlns:junos="http://xml.juniper.net/junos/*/junos"
 5    xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
 6    xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
 7    <xsl:import href="../import/junos.xsl"/>

 8    <xsl:template match="configuration">
 9      <xsl:for-each select="interfaces/interface[starts-with(name, 'so-') \
         and unit/family/inet]">
10        <transient-change>
11          <interfaces>
12            <interface>
13              <name><xsl:value-of select="name"/></name>
14              <encapsulation>ppp</encapsulation>
15            </interface>
16          </interfaces>
17        </transient-change>
18      </xsl:for-each>
19    </xsl:template>
20  </xsl:stylesheet>
```

Lines 1 through 8 are boilerplate as described in "Required Boilerplate for Commit Scripts" on page 133 and are omitted here for brevity.

Line 9 is an **<xsl:for-each>** programming instruction that examines each interface node whose names starts with **so-** and that has **family inet** enabled on any logical unit. (It appears here on two lines only for brevity.)

```
 9      <xsl:for-each select="interfaces/interface[starts-with(name, 'so-') \
         and unit/family/inet]">
```

Line 10 is the open tag for a transient change. The possible contents of the **<transient-change>** element are the same as the contents of the **<configuration>** tag element in the Junos XML protocol operation **<load-configuration>**.

```
10        <transient-change>
```

Lines 11 through 16 represent the content of the transient change. The encapsulation is set to **ppp**.

```
11          <interfaces>
12            <interface>
13              <name><xsl:value-of select="name"/></name>
14              <encapsulation>ppp</encapsulation>
15            </interface>
16          </interfaces>
```

Lines 17 through 19 close all open tags in this template.

```
17          </transient-change>
18        </xsl:for-each>
19      </xsl:template>
```

Line 20 closes the style sheet and the commit script.

```
20  </xsl:stylesheet>
```

Applying a Change to ISO-Enabled Interfaces

The following sample script ensures that interfaces that are enabled for an International Organization for Standardization (ISO) protocol also have MPLS enabled and are included at the **[edit protocols mpls interface]** hierarchy level. For a SLAX version of this example, see "Example: Controlling IS-IS and MPLS Interfaces" on page 221.

```
 1  <?xml version="1.0"?>
 2  <xsl:stylesheet version="1.0"
 3    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 4    xmlns:junos="http://xml.juniper.net/junos/*/junos"
 5    xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
 6    xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
 7    <xsl:import href="../import/junos.xsl"/>

 8      <xsl:template match="configuration">
 9        <xsl:variable name="mpls" select="protocols/mpls"/>
10        <xsl:for-each select="interfaces/interface/unit[family/iso]">
11        <xsl:variable name="ifname" select="concat(../name, '.', name)"/>
12        <xsl:if test="not(family/mpls)">
13          <xsl:call-template name="jcs:emit-change">
14            <xsl:with-param name="message">
15              <xsl:text>
16                Adding 'family mpls' to ISO-enabled interface
17              </xsl:text>
18            </xsl:with-param>
19            <xsl:with-param name="content">
20              <family>
21                <mpls/>
22              </family>
23            </xsl:with-param>
24          </xsl:call-template>
25        </xsl:if>
26        <xsl:if test="$mpls and not($mpls/interface[name = $ifname])">
27          <xsl:call-template name="jcs:emit-change">
28            <xsl:with-param name="message">
29              <xsl:text>Adding ISO-enabled interface </xsl:text>
30              <xsl:value-of select="$ifname"/>
31              <xsl:text> to [protocols mpls]</xsl:text>
32            </xsl:with-param>
33            <xsl:with-param name="dot" select="$mpls"/>
34            <xsl:with-param name="content">
35              <interface>
36                <name>
37                  <xsl:value-of select="$ifname"/>
38                </name>
39              </interface>
```

```
40              </xsl:with-param>
41            </xsl:call-template>
42          </xsl:if>
43        </xsl:for-each>
44      </xsl:template>
45  </xsl:stylesheet>
```

Lines 1 through 8 are boilerplate as described in "Required Boilerplate for Commit Scripts" on page 133 and are omitted here for brevity.

Line 9 saves a reference to the **[edit protocols mpls]** hierarchy level so that it can be referenced in the following **for-each** loop.

```
9           <xsl:variable name="mpls" select="protocols/mpls"/>
```

Line 10 examines each interface unit (logical interface) on which ISO is enabled. The **select** stops at the **unit**, but the predicate limits the selection to only those units that contain an **<iso>** element nested under a **<family>** element.

```
10          <xsl:for-each select="interfaces/interface/unit[family/iso]">
```

Line 11 builds the interface name in a variable. First, the **name** attribute of the variable declaration is set to **ifname**. In the Junos OS, an interface name is the concatenation of the device name, a period, and the unit number. At this point in the script, the context node is the unit number, because Line 10 changes the context to interfaces/interface/unit. The **../name** refers to the **<name>** element of the parent node of the context node, which is the device name (***type-fpc/pic/port***). The "**name**" token in the XPath expression refers to the **<name>** element of the context node, which is the unit number (***unit-number***). After the concatenation is performed, the XPath expression in Line 11 resolves to ***type-fpc/pic/port.unit-number***. As the **<xsl:for-each>** instruction in Line 10 traverses the hierarchy and locates ISO-enabled interfaces, the interface names are recursively stored in the **ifname** variable.

```
11          <xsl:variable name="ifname" select="concat(../name, '.', name)"/>
```

Line 12 evaluates as true for each ISO-enabled interface that does not have MPLS enabled.

```
12          <xsl:if test="not(family/mpls)">
```

Line 13 calls the **<jcs:emit-change>** template, which is a helper or convenience template in the junos.xsl file. This template is discussed in **jcs:emit-change Template**.

```
13            <xsl:call-template name="jcs:emit-change">
```

Lines 14 through 18 use the **message** parameter from the **<jcs:emit-change>** template. The message parameter is a shortcut you can use instead of explicitly including the **<warning>**, **<edit-path>**, and **<statement>** elements.

```
14              <xsl:with-param name="message">
15                <xsl:text>
16                  Adding 'family mpls' to ISO-enabled interface
17                </xsl:text>
18              </xsl:with-param>
```

Lines 19 through 23 use the **content** parameter from the **<jcs:emit-change>** template. The **content** parameter specifies the change to make, relative to the current context node.

```
19              <xsl:with-param name="content">
20                <family>
```

```
21              <mpls/>
22            </family>
23          </xsl:with-param>
```

Lines 24 and 25 close the tags opened in Lines 13 and 12, respectively.

```
24          </xsl:call-template>
25        </xsl:if>
```

Line 26 tests whether MPLS is already enabled and if this interface is not configured at the **[edit protocols mpls interface]** hierarchy level.

```
26        <xsl:if test="$mpls and not($mpls/interface[name = $ifname])">
```

Lines 27 through 41 contain another invocation of the **<jcs:emit-change>** template. In this invocation, the interface is added at the **[edit protocols mpls interface]** hierarchy level.

```
27            <xsl:call-template name="jcs:emit-change">
28              <xsl:with-param name="message">
29                <xsl:text>Adding ISO-enabled interface </xsl:text>
30                <xsl:value-of select="$ifname"/>
31                <xsl:text> to [edit protocols mpls]</xsl:text>
32              </xsl:with-param>
33              <xsl:with-param name="dot" select="$mpls"/>
34              <xsl:with-param name="content">
35                <interface>
36                <name>
37                  <xsl:value-of select="$ifname"/>
38                </name>
39              </interface>
40            </xsl:with-param>
41          </xsl:call-template>
```

Lines 42 through 45 close all open elements.

```
42        </xsl:if>
43      </xsl:for-each>
44    </xsl:template>
45  </xsl:stylesheet>
```

CHAPTER 10

Writing Commit Scripts That Generate a Custom Warning, Error, or System Log Message

This chapter discusses the following topics:

Overview of Generating Custom Warning, Error, and System Log Messages

You can use a commit script to specify configuration rules that you always want to enforce. If a rule is broken, the commit script can emit a warning, error, or system log message.

In the Junos OS command-line interface (CLI), warning messages are emitted during commit operations to alert you that the configuration is not complete or contains a syntax error. If a custom configuration rule is broken, a custom warning message notifies you about the problem. The commit script causes the warning message to be passed back to the Junos OS CLI or to a Junos XML protocol client application. Unlike error messages, warning messages do not cause the commit operation to fail, so they are used for configuration problems that do not affect network traffic. A warning is best used as a response to configuration settings that do not adhere to recommended practices. An example of this type of configuration setting might be assignment of the same user ID to different users.

Alternatively, you can generate a custom warning message for a serious configuration problem, and specify an automatic configuration change that rectifies the problem. For more information about the use of warning messages in conjunction with automatic configuration changes, see “Overview of Generating Persistent or Transient Configuration Changes” on page 155.

Unlike warning messages, a custom error message causes the commit operation to fail and notifies the user about the configuration problem. The commit script causes the error message to be passed back to the Junos OS CLI or to a Junos XML protocol client

application. Because error messages cause the commit operation to fail, they are used for problems that affect network traffic. An error message is best used as a response to configuration settings that you want to disallow—for example, when required statements are omitted from the configuration.

The Junos OS generates system log messages (also called syslog messages) to record events that occur on the device, including the following:

- Routine operations, such as creation of an OSPF protocol adjacency or a user login into the configuration database
- Failure and error conditions, such as failure to access a configuration file or unexpected closure of a connection to a child or peer process
- Emergency or critical conditions, such as device power-down due to excessive temperature

Each system log message identifies the Junos OS process that generated the message and briefly describes the operation or error that occurred. The *Junos OS System Log Messages Reference* provides more detailed information about system log messages.

With commit scripts, you can cause custom system log messages to be generated in response to particular events that you define. For example, if a configuration rule is broken, a custom message can be generated to record this occurrence. If the commit script corrects the configuration, a custom message can indicate that corrective action was taken.

Generating a Custom Warning, Error, or System Log Message

To generate a custom warning, error, or system log message, follow these steps:

1. At the start of the script, include the Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) boilerplate from "Required Boilerplate for Commit Scripts" on page 133. It is reproduced here for convenience:

XSLT Boilerplate

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <!-- ... insert your code here ... -->
  </xsl:template>
</xsl:stylesheet>
```

SLAX Boilerplate

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
```

```
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  /*
   * insert your code here
  */
}
```

2. At the position indicated by the comment "***Insert your code here***," include one or more XSLT programming instructions or their SLAX equivalents. For detailed information, see "Summary of XPath and XSLT Constructs" on page 95 and "Summary of SLAX Statements" on page 109. Commonly used XSLT constructs include the following:

   - **<xsl:choose> <xsl:when> <xsl:otherwise>**—Conditional construct that causes different instructions to be processed in different circumstances. The **<xsl:choose>** instruction contains one or more **<xsl:when>** elements, each of which tests an XPath expression. If the test evaluates as true, the XSLT processor executes the instructions in the **<xsl:when>** element. The XSLT processor processes only the instructions contained in the first **<xsl:when>** element whose **test** attribute evaluates as true. If none of the **<xsl:when>** elements' **test** attributes evaluate as true, the content of the **<xsl:otherwise>** element, if there is one, is processed.

   - **<xsl:for-each select="*xpath-expression*">**—Programming instruction that tells the XSLT processor to gather together a set of nodes and process them one by one. The nodes are selected by the Extensible Markup Language (XML) Path Language (XPath) expression in the **select** attribute. Each of the nodes is then processed according to the instructions contained in the **<xsl:for-each>** instruction. Code inside an **<xsl:for-each>** instruction is evaluated recursively for each node that matches the XPath expression. The context is moved to the node during each pass.

   - **<xsl:if test="*xpath-expression*">**—Conditional construct that causes instructions to be processed if the XPath expression in the **test** attribute evaluates to **true**.

     For example, the following programming instruction evaluates as true when the **source-route** statement is not included at the **[edit chassis]** hierarchy level:

     ```
     <xsl:if test="not(chassis/source-route)">
     ```

     In SLAX, the **if** construct looks like this:

     ```
     if (not(chassis/source-route))
     ```

   For more information about how to use programming instructions, including examples and pseudocode, see "XSLT Programming Instructions Overview" on page 30. For information about writing scripts in SLAX instead of XSLT, see "SLAX Overview" on page 35.

3. Include a **<xnm:warning>**, **<xnm:error>**, or **<syslog>** element with a **<message>** child element that specifies the content of the message.

   For warning and error messages, you can include several other child elements, such as the **<jcs:edit-path>** and **<jcs:statement>** templates, which cause the warning or

error message to include the relevant configuration hierarchy and statement information, as shown in the following examples.

This **<xnm:warning>** element:

```
<xnm:warning>
  <xsl:call-template name="jcs:edit-path">
    <xsl:with-param name="dot" select="chassis"/>
  </xsl:call-template>
  <message>IP source-route processing is not enabled.</message>
</xnm:warning>
```

emits this output when you issue the **commit** command:

```
[edit]
user@host# commit
  [edit chassis]
      warning: IP source-route processing is not enabled.
  commit complete
```

This **<xnm:error>** element:

```
<xnm:error>
  <xsl:call-template name="jcs:edit-path"/>
  <xsl:call-template name="jcs:statement"/>
  <message>Missing a description for this T1 interface.</message>
</xnm:error>
```

emits this output when you issue the **commit** command:

```
[edit]
user@host# commit
  [edit interfaces interface t1-0/0/0]
      'interface t1-0/0/0;'
      Missing a description for this T1 interface.
  error: 1 error reported by commit scripts
  error: commit script failure
```

NOTE: If you are including a warning message in conjunction with a script-generated configuration change, you can generate the warning by including the message parameter with the <jcs:emit-change> template. The message parameter causes the <jcs:emit-change> template to call the <xnm:warning> template, which sends a warning notification to the CLI. (For more information, see “Overview of Generating Persistent or Transient Configuration Changes” on page 155.)

For system log messages, the only supported child element is **<message>**:

```
<syslog>
  <message>syslog-string</message>
</syslog>
```

For a description of all the XSLT tags and attributes you can include, see “Tag Elements to Use When Generating Messages” on page 145.

For SLAX versions of these constructs, see “Example: Generating a Custom Warning Message” on page 147, “Example: Generating a Custom Error Message” on page 149, and “Example: Generating a Custom System Log Message” on page 152.

4. Save the script with a meaningful name.
5. Copy the script to either the /var/db/scripts/commit directory on the hard drive or the /config/scripts/commit directory on the flash drive. For information about setting the storage location for commit scripts, see “Storing Commit Scripts in Flash Memory” on page 189.

   If the device has dual Routing Engines and you want the script to take effect on both of them, you must copy the script to the /var/db/scripts/commit or the /config/scripts/commit directory on both Routing Engines. The **commit synchronize** command does not copy scripts between Routing Engines.
6. Enable the script by including the **file** statement at the **[edit system scripts commit]** hierarchy level:

   [edit system scripts commit]
   file *filename*;

   where ***filename*** is the name of the script.

Tag Elements to Use When Generating Messages

Table 12 on page 145 summarizes the tag elements that you can include in a custom warning, error, or system log message. For examples of how to supply data values within a script, see “Examples: Generating Custom Warning, Error, and System Log Messages” on page 147 and “Commit Script Examples” on page 201. (For detailed information about element hierarchy, see “Summary of Junos XML and XSLT Tag Elements Used in Commit Scripts” on page 287.)

Table 12: Tags and Attributes for Creating Custom Warning, Error, and System Log Messages

Data Item, XML Element, or Attribute	Required or Supported	Description
Container Tags and Attributes		
<syslog>	Required for system log messages	Indicates that a system log message is going to be recorded.
<xnm:error>	Required for error messages	Indicates that the server has encountered a problem while processing the client application’s request.
<xnm:warning>	Required for warning messages	Indicates that the server has encountered a problem while processing the client application’s request.
xmlns ***url***	Supported in warning and error messages	Names the XML namespace for the contents of the tag element. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

Table 12: Tags and Attributes for Creating Custom Warning, Error, and System Log Messages *(continued)*

Data Item, XML Element, or Attribute	Required or Supported	Description
xmlns:xnm *url*	Required for warning and error messages. The **xmlns:xnm** element is included in the script boilerplate, which sets the namespace globally.	Names the XML namespace for child tag elements that have the **xnm:** prefix on their names. The value is a URL of the form **http://xml.juniper.net/xnm/***version***/xnm**, where ***version*** is a string such as **1.1**.
Content Tags		
<column>	Supported in warning and error messages only	Identifies the element that caused the error by specifying its position as the number of characters after the first character in the line specified by the **<line-number>** tag element in the configuration file that was being loaded (which is named in the **<filename>** tag element). We recommend combining the **<column>** tag with the **<line-number>** and **<filename>** tags.
<database-status-information>	Supported in error messages only	Provides information about the users currently editing the configuration.
<edit-path>	Supported in warning and error messages only	Specifies the level in the configuration hierarchy where the problem occurred, using the CLI configuration mode banner. We recommend combining the **<edit-path>** tag with the **<statement>** tag.
<filename>	Supported in warning and error messages only	Names the configuration file that was being loaded.
<line-number>	Supported in warning and error messages only	Specifies the line number where the error occurred in the configuration file that was being loaded, which is named by the **<filename>** tag element. We recommend combining the **<line-number>** tag with the **<column>** and **<filename>** tags.
<message>	Required in warning, error, and system log messages	Describes the warning, error, or system log message in a natural-language text string.
<parse/>	Supported in error messages only	Indicates that there was a syntactic error in the request submitted by the client application.
<reason>	Supported in warning and error messages only	Describes the reason for the warning or error message.
<re-name>	Supported in warning and error messages only	Names the Routing Engine on which the process named by the **<source-daemon>** tag element is running.
<source-daemon>	Supported in warning and error messages only	Names the Junos OS module that was processing the request in which the warning or error message occurred.
<statement>	Supported in warning and error messages only	Specifies the configuration statement in effect when the problem occurred. We recommend combining the **<statement>** tag with the **<edit-path>** tag.

Table 12: Tags and Attributes for Creating Custom Warning, Error, and System Log Messages *(continued)*

Data Item, XML Element, or Attribute	Required or Supported	Description
<token>	Supported in warning and error messages only	Names the element in the request that caused the warning or error message.
<xsl:call-template name="jcs:edit-path">	Supported in warning and error messages only	Emits an **<edit-path>** element, which specifies the CLI configuration mode edit path in effect when the warning or error was generated. If the problem is not at the current position in the XML hierarchy, you can alter the edit path by passing the **dot** parameter. For example, **<xsl:param name="dot" select="system/ports/console"/>** changes the edit path to **[edit system ports console]**.
<xsl:call-template name="jcs:statement">	Supported in warning and error messages only	Emits a **<statement>** element, which describes the configuration statement in effect when the warning or error was generated. If the problem is not at the current position in the XML hierarchy, you can alter the statement by passing the **dot** parameter. For example, **<xsl:with-param name="dot" select="system/ports/console/type"/>** changes the statement to **type**.

Examples: Generating Custom Warning, Error, and System Log Messages

Example: Generating a Custom Warning Message

Using a commit script, write a custom warning message that appears when the **source-route** statement is not included at the **[edit chassis]** hierarchy level. (This example is the complete script for the sample **<xnm:warning>** element used in "Generating a Custom Warning, Error, or System Log Message" on page 142.)

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:if test="not(chassis/source-route)">
      <xnm:warning>
        <xsl:call-template name="jcs:edit-path">
          <xsl:with-param name="dot" select="chassis"/>
        </xsl:call-template>
```

```
            <message>IP source-route processing is not enabled.</message>
          </xnm:warning>
        </xsl:if>
      </xsl:template>
    </xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  if (not(chassis/source-route)) {
    <xnm:warning> {
      call jcs:edit-path($dot = chassis);
      <message> "IP source-route processing is not enabled.";
    }
  }
}
```

Verifying the Warning Message Generated by the Commit Script

To test that a commit script generates a warning message correctly, make sure that the candidate configuration contains the condition that elicits the warning. For this example, ensure that the **source-route** statement is not included at the **[edit chassis]** hierarchy level.

To test the example in this topic, perform the following steps:

1. Copy the XSLT script from the preceding section into a text file named source-route.xsl.
2. Copy the source-route.xsl file to the /var/db/scripts/commit directory on the device hard drive or the /config/scripts/commit directory on the flash drive.
3. Include the **file source-route.xsl** statement at the **[edit system scripts commit]** hierarchy level:

   ```
   user@host> edit
   [edit]
   user@host# set system scripts commit file source-route.xsl
   ```

4. If the **source-route** statement is included at the **[edit chassis]** hierarchy level, issue the **delete chassis source-route** configuration mode command:

   ```
   [edit]
   user@host# delete chassis source-route
   ```

5. Issue the **commit** command. The following output appears:

   ```
   [edit]
   user@host# commit
   [edit chassis]
     warning: IP source-route processing is not enabled.
   commit complete
   ```

To display the XML-formatted version of the warning message, issue the **commit check | display xml** command:

```
[edit]

user@host# commit check | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/junos">
    <commit-results>
        <routing-engine junos:style="normal">
            <name>re0</name>
            <xnm:warning>
                <edit-path>
                    [edit chassis]
                </edit-path>
                <message>
                    IP source-route processing is not enabled.
                </message>
            </xnm:warning>
            <commit-check-success/>
        </routing-engine>
    </commit-results>
</rpc-reply>
```

To display a detailed trace of commit script processing, issue the **commit check | display detail** command:

```
[edit]

user@host# commit check | display detail
2009-06-15 14:40:29 PDT: reading commit script configuration
2009-06-15 14:40:29 PDT: testing commit script configuration
2009-06-15 14:40:29 PDT: opening commit script
'/var/db/scripts/commit/source-route-warning.xsl'
2009-06-15 14:40:29 PDT: reading commit script 'source-route-warning.xsl'
2009-06-15 14:40:29 PDT: running commit script 'source-route-warning.xsl'
2009-06-15 14:40:29 PDT: processing commit script 'source-route-warning.xsl'
[edit chassis]
    warning: IP source-route processing is not enabled.
2009-06-15 14:40:29 PDT: no errors from source-route-warning.xsl
2009-06-15 14:40:29 PDT: saving commit script changes
2009-06-15 14:40:29 PDT: summary: changes 0, transients 0 (allowed), syslog 0
2009-06-15 14:40:29 PDT: no commit script changes
2009-06-15 14:40:29 PDT: exporting juniper.conf
2009-06-15 14:40:29 PDT: expanding groups
2009-06-15 14:40:29 PDT: finished expanding groups
2009-06-15 14:40:29 PDT: setup foreign files
2009-06-15 14:40:29 PDT: propagating foreign files
2009-06-15 14:40:30 PDT: complete foreign files
2009-06-15 14:40:30 PDT: daemons checking new configuration
configuration check succeeds
```

Example: Generating a Custom Error Message

Using a commit script, write a custom error message that appears when the **description** statement is not included at the **[edit interfaces t1-*fpc/pic/port*]** hierarchy level:

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
```

```
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:variable name="interface" select="interfaces/interface"/>
    <xsl:for-each select="$interface[starts-with(name, 't1-')]">
      <xsl:variable name="ifname" select="."/>
      <xsl:if test="not(description)">
        <xnm:error>
          <xsl:call-template name="jcs:edit-path"/>
          <xsl:call-template name="jcs:statement"/>
          <message>Missing a description for this T1 interface.</message>
        </xnm:error>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  var $interface = interfaces/interface;
  for-each ($interface[starts-with(name, 't1-')]) {
    var $ifname = .;
    if (not(description)) {
      <xnm:error> {
        call jcs:edit-path();
        call jcs:statement();
        <message> "Missing a description for this T1 interface.";
      }
    }
  }
}
```

Verifying the Error Message Generated by the Commit Script

To test that a commit script generates an error message correctly, make sure that the candidate configuration contains the condition that elicits the error. For this example, ensure that the configuration for a T1 interface does not include the **description** statement.

To test the example in this topic, perform the following steps:

1. Copy the XSLT script from the preceding section into a text file named description.xsl.
2. Copy the description.xsl file to the /var/db/scripts/commit directory on the device hard drive or the /config/scripts/commit directory on the flash drive.
3. Include the **file description.xsl** statement at the **[edit system scripts commit]** hierarchy level:

   ```
   user@host> edit
   ```

```
[edit]
user@host# set system scripts commit file description.xsl
```

4. If the configuration for every T1 interface includes the **description** statement, issue the following configuration mode commands:

```
[edit]
user@host# edit interfaces t1-0/0/1
[edit interfaces t1-0/0/1]
user@host# delete description
```

5. Issue the **commit** command. The following output appears:

```
[edit]
user@host# commit
[edit interfaces interface t1-0/0/1]
  'description'
    Missing a description for this T1 interface.
[edit interfaces interface t1-0/0/2]
  'description'
    Missing a description for this T1 interface.
error: 2 errors reported by commit scripts
error: commit script failure
```

To display the XML-formatted version of the error message, issue the **commit check | display xml** command:

```
[edit interfaces t1-0/0/1]

user@host# commit check | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/junos">
    <commit-results>
        <routing-engine junos:style="normal">
            <name>re0</name>
            <xnm:error>
                <edit-path>
                    [edit interfaces interface t1-0/0/1]
                </edit-path>
                <statement>
                    description
                </statement>
                <message>
                    Missing a description for this T1 interface.
                </message>
            </xnm:error>
            <xnm:error>
                <edit-path>
                    [edit interfaces interface t1-0/0/2]
                </edit-path>
                <statement>
                    description
                </statement>
                <message>
                    Missing a description for this T1 interface.
                </message>
            </xnm:error>
            <xnm:error xmlns="http://xml.juniper.net/xnm/1.1/xnm"
                       xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
                <message>
                    2 errors reported by commit scripts
```

```
                </message>
            </xnm:error>
            <xnm:error xmlns="http://xml.juniper.net/xnm/1.1/xnm"
          xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
                <message>
                    commit script failure
                </message>
            </xnm:error>
        </routing-engine>
    </commit-results>
    <cli>
        <banner>[edit interfaces]</banner>
    </cli>
</rpc-reply>
```

To display a detailed trace of commit script processing, issue the **commit check | display detail** command:

```
[edit interfaces t1-0/0/1]

user@host# commit check | display detail
2009-06-15 15:56:09 PDT: reading commit script configuration
2009-06-15 15:56:09 PDT: testing commit script configuration
2009-06-15 15:56:09 PDT: opening commit script '/var/db/scripts/commit/error.xsl'
2009-06-15 15:56:09 PDT: reading commit script 'error.xsl'
2009-06-15 15:56:09 PDT: running commit script 'error.xsl'
2009-06-15 15:56:09 PDT: processing commit script 'error.xsl'
[edit interfaces interface t1-0/0/1]
    'description'
        Missing a description for this T1 interface.
[edit interfaces interface t1-0/0/2]
    'description'
        Missing a description for this T1 interface.
2009-06-15 15:56:09 PDT: 2 errors from script 'error.xsl'
error: 2 errors reported by commit scripts
error: commit script failure
```

Example: Generating a Custom System Log Message

Using a commit script, write a custom system log message that appears when the **read-write** statement is not included at the **[edit snmp community *community-name* authorization]** hierarchy level.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:for-each select="snmp/community">
      <xsl:if test="not(authorization/read-write)">
        <syslog>
          <message>SNMP community does not have read-write access.
          </message>
        </syslog>
      </xsl:if>
```

```
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  for-each (snmp/community) {
    if (not(authorization/read-write)) {
      <syslog> {
        <message> "SNMP community does not have read-write access.";
      }
    }
  }
}
```

Verifying the System Log Message Generated by the Commit Script

System log messages are generated during a commit operation but not during a commit check operation. This means you cannot use the **commit check | display xml** and **commit check | display detail** configuration mode commands to verify the output of system log messages.

To test that a commit script generates a system log message correctly, make sure that the candidate configuration contains the condition that elicits the system log message. In this example, ensure that the **read-write** statement is not included at the **[edit snmp community *community-name* authorization]** hierarchy level.

To test the example in this topic, perform the following steps:

1. Copy the XSLT script from the preceding section into a text file named read-write.xsl.
2. Copy the read-write.xsl file to the /var/db/scripts/commit directory on the device hard drive or the /config/scripts/commit directory on the flash drive.
3. Include the **file read-write.xsl** statement at the **[edit system scripts commit]** hierarchy level:

   ```
   user@host> edit
   [edit]
   user@host# set system scripts commit file read-write.xsl
   ```

4. If the **read-write** statement is included at the **[edit snmp community *community-name* authorization]** hierarchy level, issue the following configuration mode command:

   ```
   [edit]
   user@host# delete snmp community community-name authorization read-write
   ```

5. Issue the following command to verify that system logging is configured to write to a file (a commonly used file name is messages):

   ```
   [edit]
   user@host# show system syslog
   ```

For information about system log configuration, see the *Junos OS System Log Messages Reference*.

6. Issue the **commit** command:

```
[edit]
user@host# commit
```

When the commit operation completes, inspect the system log file. The default directory for log files is /var/log. System log entries generated by commit scripts have the following format:

```
timestamp router-id cscript: message
```

For example:

```
Jun 3 14:34:37 router cscript: SNMP community does not have read-write access
```

CHAPTER 11

Writing Commit Scripts That Generate a Persistent or Transient Configuration Change

This chapter discusses the following topics:

Overview of Generating Persistent or Transient Configuration Changes

When a candidate configuration includes statements that you have decided must not be included in your configuration, or when the candidate omits statements that you have decided are required, commit scripts can automatically change the candidate and thereby correct the problem.

Differences Between Persistent and Transient Changes

Configuration changes made by commit scripts can be *persistent* or *transient*.

A persistent change remains in the candidate configuration and affects routing operations until you explicitly delete it, even if you subsequently remove or disable the commit script that generated the change and reissue the **commit** command. In other words, removing the commit script does not cause a persistent change to be removed from the configuration.

A transient change, in contrast, is made in the *checkout configuration* but not in the candidate configuration. The checkout configuration is the configuration database that is inspected for standard Junos OS syntax just before it is copied to become the active configuration on the device. If you subsequently remove or disable the commit script

that made the change and reissue the **commit** command, the change is no longer made to the checkout configuration and so does not affect the active configuration. In other words, removing the commit script effectively removes a transient change from the configuration.

A common use for transient changes is to eliminate the need to repeatedly configure and display well-known policies, thus allowing these policies to be enforced implicitly. For example, if MPLS must be enabled on every interface with an International Organization for Standardization (ISO) protocol enabled, the change can be transient, so that the repetitive or redundant configuration data need not be carried or displayed in the candidate configuration. Furthermore, transient changes allow you to write script instructions that apply the change only if a set of conditions is met.

Persistent and transient changes are loaded into the configuration in the same manner that the **load replace** configuration mode command loads an incoming configuration. When generating a persistent or transient change, adding the **replace="replace"** attribute to a configuration element produces the same behavior as a **replace:** tag in a **load replace** operation.

By default, the Junos OS merges the incoming configuration and the candidate configuration. New statements and hierarchies are added, and conflicting statements are overridden. When generating a persistent or transient change, if you add the **replace="replace"** attribute to a configuration element, the Junos OS replaces the existing configuration element with the incoming configuration element. If the **replace="replace"** attribute is added to a configuration element, but there is no existing element of the same name in the current configuration, the incoming configuration element is added into the configuration. Elements that do not have the**replace** attribute are merged into the configuration.

Persistent and transient changes are loaded before the standard Junos validation checks are performed. This means any configuration changes introduced by a commit script are validated for correct syntax. If the syntax is correct, the new configuration becomes the active, operational device configuration.

Persistent and transient changes have several important differences, as described in Table 13 on page 156.

Table 13: Differences Between Persistent and Transient Changes

Persistent Changes	Transient Changes
A persistent change is represented in a commit script by the **<change>** tag.	A transient change is represented in a commit script by the **<transient-change>** tag.
Another way to represent a persistent change is with the **content** parameter inside a call to the **<jcs:emit-change>** template.	Another way to represent a transient change is to use the **content** parameter and the **tag transient** parameter inside a call to the **<jcs:emit-change>** template.
The **<jcs:emit-change>** template is a helper template contained in the junos.xsl import file.	

Table 13: Differences Between Persistent and Transient Changes *(continued)*

Persistent Changes	Transient Changes
You can use persistent changes to perform any Junos XML protocol operation, such as activate, deactivate, delete, insert (reorder), comment (annotate), and replace sections of the configuration.	Like persistent changes, you can use transient changes to perform any Junos XML protocol operation. However, some Junos XML protocol operations do not make sense to use with transient changes, such as generating comments and inactive settings.
Persistent changes are always loaded during the commit process if no errors are generated by any commit scripts or by the standard Junos OS validity check.	For transient changes to be loaded, you must include the **allow-transients** statement at the **[edit system scripts commit]** hierarchy level. If you enable a commit script that generates transient changes and you do not include the **allow-transients** statement in the configuration, the CLI generates an error message and the commit operation fails. Like persistent changes, transient changes must pass the standard Junos OS validity check. You cannot use a commit script to generate the **allow-transients** statement at the **[edit system scripts commit]** hierarchy level. Rather, you must include this statement directly by using the CLI.
Persistent changes work like the **load replace** configuration mode command, and the change is added to the candidate configuration. When generating a persistent change, if you add the **replace="replace"** attribute to a configuration element, the Junos OS replaces the existing element in the candidate configuration with the incoming configuration element. If there is no existing element of the same name in the candidate configuration, the incoming configuration element is added into the configuration. Elements that do not have the **replace** attribute are merged into the configuration.	Transient changes work like the **load replace** configuration mode command, and the change is added to the checkout configuration. When generating a transient change, if you add the **replace="replace"** attribute to a configuration element, the Junos OS replaces the existing element in the checkout configuration with the incoming configuration element. If there is no existing element of the same name in the checkout configuration, the incoming configuration element is added into the configuration. Elements that do not have the **replace** attribute are merged into the configuration. Transient changes are not copied to the candidate configuration. For this reason, transient changes are not saved in the configuration if the associated commit script is deleted or deactivated.
After a persistent change is committed, the software treats it like a change you make by directly editing and committing the candidate configuration. After the persistent changes are copied to the candidate configuration, they are copied to the checkout configuration. If the changes pass the standard Junos OS validity checks, the changes are propagated to the switch, router, or security device components.	Each time a transient change is committed, the software updates the checkout configuration database. After the transient changes pass the standard Junos OS validity checks, the changes are propagated to the device components.

Table 13: Differences Between Persistent and Transient Changes *(continued)*

Persistent Changes	Transient Changes
After committing a script that causes a persistent change to be generated, you can view the persistent change by issuing the **show** configuration mode command: user@host# **show** This command displays persistent changes only, not transient changes.	After committing a script that causes a transient change to be generated, you can view the transient change by issuing the **show | display commit-scripts** configuration mode command: user@host# **show | display commit-scripts** This command displays both persistent and transient changes.
Persistent changes must conform to your custom configuration design rules as dictated by commit scripts. This does not become apparent until after a second commit operation because persistent changes are not evaluated by commit script rules on the current commit operation. The subsequent commit operation fails if the persistent changes do not conform to the rules imposed by the commit scripts configured during the first commit operation.	Transient changes are never tested by and do not need to conform to your custom rules. This is caused by the order of operations in the Junos OS commit model, which is explained in detail in "Commit Scripts and the Junos OS Commit Model" on page 130.
A persistent change remains in the configuration even if you delete, disable, or deactivate the commit script instructions that generated the change.	If you delete, disable, or deactivate the commit script instructions that generate a transient change, the change is removed from the configuration after the next commit operation. In short, if the associated instructions or the entire commit script is removed, the transient change is also removed.
As with direct CLI configuration, you can remove a persistent change by rolling back to a previous configuration that did not include the change and issuing the **commit** command. However, if you do not disable or deactivate the associated commit script, and the problem that originally caused the change to be generated still exists, the change is automatically regenerated when you issue another **commit** command.	You cannot remove a transient change by rolling back to a previous configuration.
You can alter persistent changes directly by editing the configuration using the CLI.	You cannot directly alter or delete a transient change by using the Junos OS CLI, because the change is not in the candidate configuration. To alter the contents of a transient change, you must alter the statements in the commit script that generates the transient change.

Interaction of Configuration Changes and Configuration Groups

Any configuration change you can make by directly editing the configuration using the Junos OS command-line interface (CLI) can also be generated by a commit script as a persistent or transient change. This includes values specified at a specific hierarchy level or in configuration groups. As with direct CLI configuration, values specified in the *target* override values inherited from a configuration group. The target is the statement to which you apply a configuration group by including the **apply-groups** statement.

If you define persistent or transient changes as belonging to a configuration group, the configuration groups are applied in the order you specify in the **apply-groups** statements,

which you can include at any hierarchy level except the top level. You can also disable inheritance of a configuration group by including the **apply-groups-except** statement at any hierarchy level except the top level.

CAUTION: Each commit script inspects the postinheritance view of the configuration. If a candidate configuration contains a configuration group, be careful when using a commit script to change the related target configuration, because doing so might alter the intended inheritance from the configuration group.

Also be careful when using a commit script to change a configuration group, because the configuration group might be generated by an application that performs a load replace operation on the group during each commit operation.

For more information about configuration groups, see the *Junos OS CLI User Guide*.

Tag Elements and Templates for Generating Changes

To generate changes, you can use the **<jcs:emit-change>** template, which implicitly includes **<change>** and **<transient-change>** XML elements; or you can explicitly include **<change>** and **<transient-change>** XML elements. Using the **<jcs:emit-change>** template allows you to set the hierarchical context of the change once rather than multiple times.

The **<change>** and **<transient-change>** elements are similar to the **<load-configuration>** operation defined by the Junos XML management protocol. The possible contents of the **<change>** and **<transient-change>** elements are the same as the contents of the **<configuration>** tag element used in the Junos XML protocol operation **<load-configuration>**. For complete details about the **<load-configuration>** element, see the *Junos XML Management Protocol Guide*.

Generating a Persistent or Transient Change

To generate a persistent or transient change, follow these steps:

1. At the start of the script, include the Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) boilerplate from "Required Boilerplate for Commit Scripts" on page 133. It is reproduced here for convenience:

XSLT Boilerplate

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <!-- ... Insert your code here ... -->
  </xsl:template>
```

```
</xsl:stylesheet>
```

SLAX Boilerplate

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  /*
  * Insert your code here
  */
}
```

2. At the position indicated by the comment "***Insert your code here***," include one or more XSLT programming instructions or their SLAX equivalents. For detailed information, see "Summary of XPath and XSLT Constructs" on page 95 and "Summary of SLAX Statements" on page 109. Commonly used XSLT constructs include the following.

   - **<xsl:choose> <xsl:when> <xsl:otherwise>**—Conditional construct that causes different instructions to be processed in different circumstances. The **<xsl:choose>** instruction contains one or more **<xsl:when>** elements, each of which tests an XPath expression. If the test evaluates as true, the XSLT processor executes the instructions in the **<xsl:when>** element. The XSLT processor processes only the instructions contained in the first **<xsl:when>** element whose **test** attribute evaluates as true. If none of the **<xsl:when>** elements' **test** attributes evaluate as true, the content of the **<xsl:otherwise>** element, if there is one, is processed.

   - **<xsl:for-each select="*xpath-expression*">**—Programming instruction that tells the XSLT processor to gather together a set of nodes and process them one by one. The nodes are selected by the Extensible Markup Language (XML) Path Language (XPath) expression in the **select** attribute. Each of the nodes is then processed according to the instructions contained in the **<xsl:for-each>** instruction. Code inside an **<xsl:for-each>** instruction is evaluated recursively for each node that matches the XPath expression. The context is moved to the node during each pass.

   - **<xsl:if test="*xpath-expression*">**—Conditional construct that causes instructions to be processed if the XPath expression in the **test** attribute evaluates to **true**.

   For example, the following XSLT programming instructions select each SONET/SDH interface that does not have the MPLS protocol family enabled:

   ```
   <xsl:for-each select="interfaces/interface[starts-with(name, 'so-')]/unit">
     <xsl:if test="not(family/mpls)">
   ```

   In SLAX, the **for-each** and **if** constructs look like this:

   ```
   for-each (interfaces/interface[starts-with(name, 'so-')]/unit) {
     if (not(family/mpls)) {
   ```

   For more information about how to use programming instructions, including examples and pseudocode, see "XSLT Programming Instructions Overview" on page 30. For

information about writing scripts in SLAX instead of XSLT, see “SLAX Overview” on page 35.

3. Include instructions for changing the configuration. There are two ways to generate a persistent change and two ways to generate a transient change. To generate a persistent change, you can either reference the **<jcs:emit-change>** template or include a **<change>** element. To generate a transient change, you can either reference the **<jcs:emit-change>** template and pass in the **tag** parameter with **'transient-change'** selected or include a **<transient-change>** element.

   The **<jcs:emit-change>** template allows for more efficient, less error-prone scripting because you can define the content of the change without specifying the complete XML hierarchy for the affected statement. Instead, the XML hierarchy is defined in the XPath expression contained in the script’s programming instruction.

   Consider the following examples. Both of the persistent change examples have the same result, even though they place the **unit** statement in different locations in the **<xsl:for-each>** and **<xsl:if>** programming instructions. In both cases, the script searches for SONET/SDH interfaces that do not have the MPLS protocol family enabled, adds the **family mpls** statement at the **[edit interfaces so-*fpc/pic/port* unit *logical-unit-number*]** hierarchy level, and emits a warning message stating that the configuration has been changed. Likewise, both of the transient change examples have the same result. They both set Point-to-Point Protocol (PPP) encapsulation on all SONET/SDH interface that have IP version 4 (IPv4) enabled.

Persistent Change Generated with the <jcs:emit-change> Template

In this example, the content of the persistent change (contained in the **content** parameter) is specified without including the complete XML hierarchy. Instead, the XPath expression in the **<xsl:for-each>** programming instruction sets the context for the change.

The message parameter is also included. This parameter causes the **<jcs:emit-change>** template to call the **<xnm:warning>** template, which sends a warning notification to the CLI. The message parameter automatically includes the current hierarchy information in the warning message. (For more information about the parameters available with the **<jcs:emit-change>** template, see **jcs:emit-change Template**.)

```
<xsl:for-each select="interfaces/interface[starts-with(name, 'so-')]/unit">
  <xsl:if test="not(family/mpls)">
    <xsl:call-template name="jcs:emit-change">
      <xsl:with-param name="content">
          <family>
            <mpls/>
          </family>
      </xsl:with-param>
      <xsl:with-param name="message">
        <xsl:text>Adding 'family mpls' to SONET interface.</xsl:text>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:if>
</xsl:for-each>
```

Persistent Change Generated with the <change> Element

In this example, the complete XML hierarchy leading to the affected statement must be included as child elements of the **<change>** element.

This example includes the current hierarchy information in the warning message by referencing the **<jcs:edit-path>** and **<jcs:statement>** templates. For more information about warning messages, see "Overview of Generating Custom Warning, Error, and System Log Messages" on page 141.

```
<xsl:for-each select="interfaces/interface[starts-with(name, 'so-')]">
  <xsl:if test="not(unit/family/mpls)">
    <change>
      <interfaces>
        <interface>
          <name><xsl:value-of select="name"/></name>
          <unit>
            <name><xsl:value-of select="unit/name"/></name>
            <family>
              <mpls/>
            </family>
          </unit>
        </interface>
      </interfaces>
    </change>
    <xnm:warning>
      <xsl:call-template name="jcs:edit-path"/>
      <xsl:call-template name="jcs:statement">
        <xsl:with-param name="dot" select="unit/name"/>
      </xsl:call-template>
      <message>Adding 'family mpls' to SONET interface.</message>
    </xnm:warning>
  </xsl:if>
</xsl:for-each>
```

Transient Change Generated with the <jcs:emit-change> Template

In this example, the content of the transient change (contained in the **content** parameter) is specified without including the complete XML hierarchy. Instead, the XPath expression in the **<xsl:for-each>** programming instruction sets the context of the change. The **and** operator in the XPath expression means both operands are **true** when converted to Booleans; the second operand is not evaluated if the first operand is **false**.

The tag parameter is included with '**transient-change**' selected. Without the **tag** parameter, the **<jcs:emit-change>** template generates a persistent change by default. (For more information about the parameters available with the **<jcs:emit-change>** template, see "Junos Named Templates in the jcs Namespace" on page 82.)

```
<xsl:for-each select="interfaces/interface[starts-with(name, 'so-') \
              and unit/family/inet]">
  <xsl:call-template name="jcs:emit-change">
    <xsl:with-param name="tag" select="'transient-change'"/>
    <xsl:with-param name="content">
      <encapsulation>ppp</encapsulation>
    </xsl:with-param>
  </xsl:call-template>
```



```
</xsl:for-each>
```

Transient Change Generated with the <transient-change> Element

In this example, the complete XML hierarchy leading to the affected statement must be included as child elements of the **<transient-change>** element.

```
<xsl:for-each select="interfaces/interface[starts-with(name, 'so-')\
            and unit/family/inet]">
  <transient-change>
    <interfaces>
      <interface>
        <name><xsl:value-of select="name"/></name>
        <encapsulation>ppp</encapsulation>
      </interface>
    </interfaces>
  </transient-change>
</xsl:for-each>
```

4. Save the script with a meaningful name.
5. Copy the script to either the /var/db/scripts/commit directory on the device hard drive or the /config/scripts/commit directory on the flash drive. For information about setting the storage location for commit scripts, see "Storing Commit Scripts in Flash Memory" on page 189.

   If the device has dual Routing Engines and you want the script to take effect on both of them, you must copy the script to the /var/db/scripts/commit or the /config/scripts/commit directory on both Routing Engines. The **commit synchronize** command does not copy scripts between Routing Engines.
6. Enable the script by including the **file** statement at the **[edit system scripts commit]** hierarchy level:

   ```
   [edit system scripts commit]
   file filename;
   ```

   where ***filename*** is the name you assigned in Step 4.
7. If the script makes transient changes, include the **allow-transients** statement at the **[edit system scripts commit]** hierarchy level:

   ```
   [edit system scripts commit]
   allow-transients;
   ```

If all the commit scripts run without errors, any transient changes are loaded into the checkout configuration, but not to the candidate configuration. Any persistent changes are loaded into the candidate configuration. The commit process then continues by validating the configuration and propagating changes to the affected processes on the device.

To display the configuration with both persistent and transient changes applied, issue the **show | display commit-scripts** configuration mode command:

```
[edit]
user@host# show | display commit-scripts
```

To display the configuration with only persistent changes applied, issue the **show | display commit-scripts no-transients** configuration mode command:

```
[edit]
user@host# show | display commit-scripts no-transients
```

Persistent and transient changes are loaded into the configuration in the same manner that the **load replace** configuration mode command loads an incoming configuration. When generating a persistent or transient change, adding the **replace="replace"** attribute to a configuration element produces the same behavior as a **replace:** tag in a **load replace** operation. Both persistent and transient changes are loaded into the configuration with the **load replace** behavior, but persistent changes are loaded into the candidate configuration and transient changes are loaded into the checkout configuration.

Removing a Persistent or Transient Change

After a commit script changes the configuration, you can remove the change and return the configuration to its previous state.

For persistent changes only, you can undo the configuration change by issuing the **delete**, **deactivate**, or **rollback** configuration mode command and committing the configuration. For both persistent and transient changes, you must remove, delete, or deactivate the associated commit script, or else the commit script regenerates the change during a subsequent commit operation.

Deleting the **file** ***filename*** statement from the configuration effectively "unconfigures" the functionality associated with the corresponding commit script. Deactivating the statement adds the **inactive:** tag to the statement, effectively commenting out the statement from the configuration. Statements marked as inactive do not take effect when you issue the **commit** command.

To reverse the effect of a commit script and prevent the script from running again, perform the following steps:

1. For persistent changes only, delete or deactivate the statement that was added by the commit script:

   ```
   [edit]
   user@host# delete (statement | identifier)
   - OR –
   user@host# deactivate (statement | identifier)
   ```

   Alternatively, you can roll back the configuration to a candidate that does not contain the statement.

   ```
   [edit]
   user@host# rollback number
   ```

2. Either delete or deactivate the commit script, or remove or comment out the section of code that generates the unwanted change. To delete or deactivate the script, issue one of the following commands.

   ```
   [edit]
   user@host# delete system scripts commit file filename
   - OR –
   ```

```
user@host# deactivate system scripts commit file filename
```

3. Issue the **commit** command:

```
[edit]
user@host# commit
```

4. If you are deleting the reference to the script from the configuration, you can also remove the file from commit scripts storage directory (either /var/db/scripts/commit on the hard drive or /config/scripts/commit on the flash drive; for information about setting the storage location for commit scripts, see "Storing Commit Scripts in Flash Memory" on page 189.) To do this, exit configuration mode and issue the **file delete** operational mode command:

```
[edit]
user@host# exit

user@host> file delete /var/db/scripts/commit/filename
- OR –
user@host> file delete /config/scripts/commit/filename
```

Tag Elements to Use When Generating Persistent and Transient Changes

Table 14 on page 165 describes the data that you can include in the **<change>** tag element in a commit script. To see how data values are supplied within a script, see "Examples: Generating Persistent and Transient Changes" on page 166 and "Commit Script Examples" on page 201. (For detailed information about element hierarchy, see "Summary of Junos XML and XSLT Tag Elements Used in Commit Scripts" on page 287.)

Table 14: Tags and Attributes for Creating Configuration Changes

Data Item, XML Element, or Attribute	Description
Container Tags	
<change>	Request that the Junos XML protocol server load configuration data into the candidate configuration.
<transient-change>	Request that the Junos XML protocol server load configuration data into the configuration.
Content Tags	
<jcs:emit-change>	This is a template in the file junos.xsl. This template converts the contents of the **<xsl:with-param>** element into a **<change>** request.
<xsl:with-param name="content">	You use the **content** parameter with the **<jcs:emit-change>** template. It allows you to include the content of the change, relative to **dot**.

Table 14: Tags and Attributes for Creating Configuration Changes *(continued)*

Data Item, XML Element, or Attribute	Description
<xsl:with-param name="tag" select="'transient-change'"/>	Convert the contents of the **content** parameter into a **<transient-change>** request. You use the **tag** parameter with the **<jcs:emit-change>** template. By default, the **<jcs:emit-change>** template converts the contents of the **content** parameter into a **<change>** (persistent change) request.

Examples: Generating Persistent and Transient Changes

This section is organized as follows:

Example: Generating a Persistent Change

If you do not explicitly configure the MPLS protocol family on an interface, the interface is not enabled for MPLS applications. This example generates a persistent change that adds the **family mpls** statement in the configuration of SONET/SDH interfaces when the statement is not already included in the configuration.

The persistent change is generated by the **<jcs:emit-change>** template, which is a helper template contained in the junos.xsl import file. The **content** parameter of the **<jcs:emit-change>** template includes the configuration statements to be added as a persistent change. The **message** parameter of the **<jcs:emit-change>** template includes the warning message to be displayed at the CLI, notifying you that the configuration has been changed.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:for-each select="interfaces/interface[starts-with(name, 'so-')]/unit">
      <xsl:if test="not(family/mpls)">
        <xsl:call-template name="jcs:emit-change">
          <xsl:with-param name="message">
            <xsl:text>Adding 'family mpls' to SONET/SDH interface.</xsl:text>
          </xsl:with-param>
          <xsl:with-param name="content">
              <family>
```

```
                    <mpls/>
                  </family>
                </xsl:with-param>
              </xsl:call-template>
            </xsl:if>
          </xsl:for-each>
        </xsl:template>
      </xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  for-each (interfaces/interface[starts-with(name, 'so-')]/unit) {
    if (not(family/mpls)) {
      call jcs:emit-change() {
        with $message = {
          expr "Adding 'family mpls' to SONET/SDH interface.";
        }
        with $content = {
          <family> {
            <mpls>;
          }
        }
      }
    }
  }
}
```

Verifying the Persistent Change Generated by the Commit Script

To test that a commit script generates a persistent change correctly, make sure that the candidate configuration contains the condition that elicits the change. For this example, ensure that the **family mpls** statement is not included at the **[edit interfaces so-*fpc/pic/port* unit *logical-unit-number*]** hierarchy level.

The sample script produces a message announcing the change it is making. To display the XML-formatted version of the message, issue the **commit check | display xml** command:

```
[edit]

user@host# commit check | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/junos">
    <commit-results>
        <routing-engine junos:style="normal">
            <name>re0</name>
            <xnm:warning>
                <edit-path>
                    [edit interfaces interface so-2/3/4 unit 0]
                </edit-path>
                <message>
                    Adding 'family mpls' to SONET interface.
                </message>
```

```
            </xnm:warning>
            <commit-check-success/>
        </routing-engine>
    </commit-results>
</rpc-reply>
```

To display a detailed trace of commit script processing, issue the **commit check | display detail** command:

```
[edit]

user@host# commit check | display detail
2009-06-17 14:17:35 PDT: reading commit script configuration
2009-06-17 14:17:35 PDT: testing commit script configuration
2009-06-17 14:17:35 PDT: opening commit script '/var/db/scripts/commit/mpls.xsl'
2009-06-17 14:17:35 PDT: reading commit script 'mpls.xsl'
2009-06-17 14:17:35 PDT: running commit script 'mpls.xsl'
2009-06-17 14:17:35 PDT: processing commit script 'mpls.xsl'
2009-06-17 14:17:35 PDT: no errors from mpls.xsl
2009-06-17 14:17:35 PDT: saving commit script changes
2009-06-17 14:17:35 PDT: summary: changes 0, transients 0 (allowed), syslog 0
2009-06-17 14:17:35 PDT: no commit script changes
2009-06-17 14:17:35 PDT: finished loading commit script changes
2009-06-17 14:17:35 PDT: exporting juniper.conf
2009-06-17 14:17:35 PDT: expanding groups
2009-06-17 14:17:35 PDT: finished expanding groups
2009-06-17 14:17:35 PDT: setup foreign files
2009-06-17 14:17:35 PDT: propagating foreign files
2009-06-17 14:17:35 PDT: complete foreign files
2009-06-17 14:17:36 PDT: daemons checking new configuration
configuration check succeeds
```

To view the configuration with the persistent change, issue the **show interfaces** configuration mode command. If the MPLS protocol family is not enabled on one or more SONET/SDH interfaces before the script runs, something similar to the following appears after the commit script runs:

```
[edit]

user@host# show interfaces
... other configured interface types ...
so-2/3/4 {
    unit 0 {
  family mpls; # Added by persistent change
    }
}
... other configured interface types ...
```

Example: Generating a Transient Change

Using a commit script, make a transient configuration change that sets PPP encapsulation on all SONET/SDH interfaces with the IPv4 protocol family enabled:

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
```

```
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:for-each select="interfaces/interface[starts-with(name, 'so-')
            and unit/family/inet]">
      <xsl:call-template name="jcs:emit-change">
        <xsl:with-param name="tag" select="'transient-change'"/>
        <xsl:with-param name="content">
          <encapsulation>ppp</encapsulation>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  for-each (interfaces/interface[starts-with(name, 'so-') and unit/family/inet]) {
    call jcs:emit-change($tag = 'transient-change') {
      with $content = {
        <encapsulation> "ppp";
      }
    }
  }
}
```

Verifying the Transient Change Generated by the Commit Script

To display a detailed trace of commit script processing, issue the **commit check | display detail** command:

```
[edit]

user@host# commit check | display detail
2009-06-15 12:07:30 PDT: reading commit script configuration
2009-06-15 12:07:30 PDT: testing commit script configuration
2009-06-15 12:07:30 PDT: opening commit script
'/var/db/scripts/commit/transient.xsl'
2009-06-15 12:07:30 PDT: reading commit script 'transient.xsl'
2009-06-15 12:07:30 PDT: running commit script 'transient.xsl'
2009-06-15 12:07:30 PDT: processing commit script 'transient.xsl'
2009-06-15 12:07:30 PDT: no errors from transient.xsl
2009-06-15 12:07:30 PDT: saving commit script changes
2009-06-15 12:07:30 PDT: summary: changes 0, transients 2 (allowed), syslog 0
2009-06-15 12:07:30 PDT: no commit script changes
2009-06-15 12:07:30 PDT: exporting juniper.conf
2009-06-15 12:07:30 PDT: loading transient changes
2009-06-15 12:07:30 PDT: loading commit script changes(transient)
2009-06-15 12:07:30 PDT: finished loading commit script changes
2009-06-15 12:07:30 PDT: expanding groups
2009-06-15 12:07:30 PDT: finished expanding groups
2009-06-15 12:07:30 PDT: setup foreign files
2009-06-15 12:07:30 PDT: propagating foreign files
```

```
2009-06-15 12:07:31 PDT: complete foreign files
2009-06-15 12:07:31 PDT: daemons checking new configuration
configuration check succeeds
```

To display the configuration with the transient change, issue the **show interfaces | display commit-scripts** configuration mode command. If there are one or more SONET/SDH interfaces with the IPv4 protocol family enabled, the output is similar to this:

```
[edit]
user@host# show interfaces | display commit-scripts
... other configured interface types ...
so-1/2/3 {
    mtu 576;
    encapsulation ppp; /* Added by transient change. */
    unit 0 {
        family inet {
            address 10.0.0.3/32;
        }
    }
}
so-1/2/4 {
    encapsulation ppp; /* Added by transient change. */
    unit 0 {
        family inet {
            address 10.0.0.4/32;
        }
    }
}
so-2/3/4 {
   encapsulation cisco-hdlc; # Not affected by this script, because IPv4 protocol
                             # family is not configured on this interface.
    unit 0 {
        family mpls;
    }
}
... other configured interface types ...
```

CHAPTER 12

Writing Commit Scripts That Create Custom Configuration Syntax with Macros

This chapter discusses the following topics:

Overview of Creating Custom Configuration Syntax with Macros

Using commit script macros, you can create a custom configuration language based on simplified syntax that is relevant to your network design. This means you can use your own aliases for frequently used configuration statements.

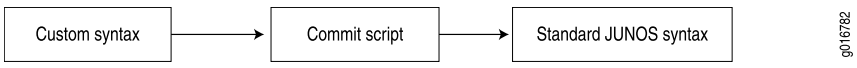
Commit scripts generally impose restrictions on Junos OS configuration and automatically correct configuration mistakes when they occur (as discussed in "Overview of Generating Persistent or Transient Configuration Changes" on page 155). However, macros are useful for an entirely different reason. Commit scripts that contain macros do not generally correct configuration mistakes, nor do they necessarily restrict configuration. Instead, they provide a way to simplify and speed configuration tasks, thereby preventing mistakes from occurring at all.

For a detailed example of how macros can save time and effort, see "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259.

How Macros Work

Your custom syntax serves as input to a commit script. The output of the commit script is standard Junos OS configuration syntax, as shown in Figure 8 on page 171. The standard Junos OS statements are added to the configuration to cause your intended operational changes.

Figure 8: Macro Input and Output



Macros use either permanent or transient change elements to expand your custom syntax into standard Junos OS configuration statements. If you use transient changes, the custom syntax appears in the candidate configuration, and the standard Junos OS syntax is copied to the checkout configuration only. If you use persistent changes, both the custom syntax and the standard Junos OS syntax appear in the candidate configuration.

This section discusses the following topics:

Creating a Custom Syntax

Macros work by locating **apply-macro** statements that you include in the candidate configuration and using the values specified in the **apply-macro** statement as parameters to a set of instructions defined in a commit script. In effect, your custom configuration syntax serves a dual purpose. The syntax allows you to simplify your configuration tasks, and it provides to the script the data necessary to generate a complex configuration.

To enter custom syntax, you include the **apply-macro** statement at any hierarchy level and specify any data that you want inside the **apply-macro** statement:

```
apply-macro macro-name {
  parameter-name parameter-value;
}
```

You can include the **apply-macro** statement at any level of the configuration hierarchy. In this sense, the **apply-macro** statement is similar to the **apply-groups** statement. Each **apply-macro** statement must be uniquely named, relative to other **apply-macro** statements at the same hierarchy level.

An **apply-macro** statement can contain a set of parameters with optional values. The corresponding commit script can refer to the macro name, its parameters, or the parameters' values. When the script inspects the configuration and finds the data, the script performs the actions specified by a persistent or transient change element.

For example, given the following configuration stanza, you can write script instructions to generate a standard configuration based on the name of the parameter:

```
protocols {
  mpls {
    apply-macro blue-type-lsp {
      color blue;
    }
  }
}
```

The following **<xsl:for-each>** programming instruction finds **apply-macro** statements at the **[edit protocols mpls]** hierarchy level that contain a parameter named **color**:

```
<xsl:for-each select="protocols/mpls/apply-macro[data/name = 'color']">
```

The following instruction creates a variable named **color** and assigns to the variable the value of the **color** parameter, which in this case is **blue**:

```
<xsl:variable name="color" select="data[name = 'color']/value"/>
```

The following instruction adds the **admin-groups** statement to the configuration and assigns the value of the **$color** variable to the group name:

```
<transient-change>
  <protocols>
    <mpls>
      <admin-groups>
        <name>
          <xsl:value-of select="$color"/>
        </name>
      </admin-groups>
    </mpls>
  </protocols>
</transient-change>
```

The resulting configuration statements are as follows:

```
protocols {
  mpls {
    admin-groups {
      blue;
    }
  }
}
```

<data> Element

In the XML rendering of the custom syntax within an **apply-macro** statement, parameters and their values are contained in **<name>** and **<value>** elements, respectively. The **<name>** and **<value>** elements are sibling children of the **<data>** element. For example, the **apply-macro blue-type-lsp** statement contains six parameters, as follows:

```
[edit protocols mpls]
apply-macro blue-type-lsp {
  10.1.1.1;
  10.2.2.2;
  10.3.3.3;
  10.4.4.4;
  color blue;
  group-value 0;
}
```

The parameters and values are rendered in Junos XML tag elements as follows:

```
[edit protocols mpls]
user@host# show | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/junos">
  <configuration>
    <protocols>
      <mpls>
        <apply-macro>
          <name>blue-type-lsp</name>
          <data>
```

```
            <name>10.1.1.1</name>
          </data>
          <data>
            <name>10.2.2.2</name>
          </data>
          <data>
            <name>10.3.3.3</name>
          </data>
          <data>
            <name>10.4.4.4</name>
          </data>
          <data>
            <name>color</name>
            <value>blue</value>
          </data>
          <data>
            <name>group-value</name>
            <value>0</value>
          </data>
        </apply-macro>
      </mpls>
    </protocols>
  </configuration>
</rpc-reply>
```

When you write commit script macros, referring to the **<data>**, **<name>**, and **<value>** elements enables you to extract and manipulate the parameters contained in **apply-macro** statements. For example, in the following **select** attribute, the XPath expression extracts the text contained in the **<value>** element that is a child of a **<data>** element that also contains a **<name>** child element with the text **color**. The variable declaration assigns the text of the **<value>** element to a variable named **$color**.

```
<xsl:variable name="color" select="data[name = 'color']/value"/>
```

Expanding the Custom Syntax

In the corresponding commit script, you include one or more XSLT or SLAX programming instructions that inspect the configuration for the **apply-macro** statement at a specified hierarchy level. Optionally, you can use the **data/name** expression to select a parameter in the **apply-macro** statement:

```
<xsl:for-each select="xpath-expression/apply-macro[data/name = 'parameter-name']">
```

For example, the following XSLT programming instruction selects every **apply-macro** statement that contains the **color** parameter and that appears at the **[edit protocols mpls]** hierarchy level:

```
<xsl:for-each select="protocols/mpls/apply-macro[data/name = 'color']">
```

The SLAX equivalent is:

```
for-each (protocols/mpls/apply-macro[data/name = 'color'])
```

When expanding macros, a particularly useful programming instruction is the **<xsl:value-of>** instruction. This instruction selects a parameter value and uses it to build option values for Junos OS statements. For example, the following instruction concatenates the value of the **$color** variable, the text **-lsp-**, and the current context node (represented by "**.**") to build a name for an LSP.

```
<label-switched-path>
  <name>
    <xsl:value-of select="concat($color, '-lsp-', .)"/>
  </name>
</label-switched-path>
```

SLAX uses the underscore (_) to concatenate values:

```
<label-switched-path> {
  <name> $color _ '-lsp-' _ .;
```

When the script includes instructions to find the necessary data, you can provide content for a transient change that uses the data to construct a standard Junos OS configuration.

The following transient change creates an administration group and adds the **label-switched-path** statement to the configuration. The label-switched path is assigned a name that concatenates the value of the **$color** variable, the text **-lsp-**, and the currently selected IP address represented by the period ("**.**"). The transient change also adds the **to** statement and assigns the currently selected IP address. Finally, the transient change adds the **admin-group include-any** statement and assigns the value of the **$color** variable.

```
<transient-change>
  <protocols>
    <mpls>
      <admin-groups>
        <name><xsl:value-of select="$color"/></name>
        <group-value><xsl:value-of select="$group-value"/></group-value>
      </admin-groups>
      <xsl:for-each select="data[not(value)]/name">
        <label-switched-path>
          <name><xsl:value-of select="concat($color, '-lsp-', .)"/></name>
          <to><xsl:value-of select="."/></to>
          <admin-group>
            <include-any><xsl:value-of select="$color"/></include-any>
          </admin-group>
        </label-switched-path>
      </xsl:for-each>
    </mpls>
  </protocols>
</transient-change>
```

The SLAX equivalent is:

```
<transient-change> {
  <protocols> {
    <mpls> {
      <admin-groups> {
        <name> $color;
        <group-value> $group-value;
      }
      for-each (data[not(value)]/name) {
```

```
            <label-switched-path> {
              <name> $color _ '-lsp-' _ .;
              <to> .;
              <admin-group> {
                <include-any> $color;
              }
            }
          }
        }
      }
    }
```

NOTE: The example shown here is partial. For a full example, see "Example: Creating Custom Configuration Syntax with Macros" on page 178.

After committing the configuration, the script runs, and the resulting full configuration looks like this:

```
[edit]
protocols {
  mpls {
    label-switched-path blue-lsp-10.1.1.1 {
      to 10.1.1.1;
      admin-group include-any blue;
    }
    label-switched-path blue-lsp-10.2.2.2 {
      to 10.2.2.2;
      admin-group include-any blue;
    }
    label-switched-path blue-lsp-10.3.3.3 {
      to 10.3.3.3;
      admin-group include-any blue;
    }
    label-switched-path blue-lsp-10.4.4.4 {
      to 10.4.4.4;
      admin-group include-any blue;
    }
  }
}
```

The previous example demonstrates how you can use a simplified custom syntax to configure label-switched paths (LSPs). If your network design requires a large number of LSPs to be configured, using a commit script macro can save time, ensure consistency, and prevent configuration errors.

Other Ways to Use Macros

The example discussed in "Creating a Custom Syntax" on page 172 shows a macro that uses transient changes to create the intended operational impact. Alternatively, you can create a commit script that uses persistent changes to add the standard Junos OS statements to the candidate configuration and delete your custom syntax entirely. This way, a network operator who might be unfamiliar with your custom syntax can view the configuration file and see the full configuration rendered as standard Junos OS statements.

Still, because the commit script macro remains in effect, you can quickly and easily create a complex configuration using your custom syntax.

In addition to the type of application discussed in “Creating a Custom Syntax” on page 172, you can also use macros to prevent a commit script from performing a task. For example, a basic commit script that automatically adds MPLS configuration to interfaces can make an exception for interfaces you explicitly tag as not requiring MPLS, by testing for the presence of an **apply-macro** statement named **no-mpls**. For an example of this use of macros, see “Example: Controlling LDP Configuration” on page 230.

You can use the **apply-macro** statement as a place to store external data. The commit script does not inspect the **apply-macro** statement, so the **apply-macro** statement has no operational impact on the device, but the data can be carried in the configuration file to be used by external applications.

Creating a Macro to Read the Custom Syntax and Generate Related Configuration Statements

By itself, the custom syntax in an **apply-macro** statement has no operational impact on the device. To give meaning to your syntax, there must be a corresponding commit script that uses the syntax as data for generating related standard Junos OS statements. To write such a script, follow these steps:

1. At the start of the script, include the Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) boilerplate from “Required Boilerplate for Commit Scripts” on page 133. It is reproduced here for convenience:

XSLT Boilerplate

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <!-- ... Insert your code here ... -->
  </xsl:template>
</xsl:stylesheet>
```

SLAX Boilerplate

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  /*
  * Insert your code here
  */
```

```
}
```

2. At the position indicated by the comment "***Insert your code here***," include XSLT programming instructions (or their SLAX equivalents) that inspect the configuration for the **apply-macro** statement at a specified hierarchy level and change the configuration to include standard Junos OS CLI syntax.

   For detailed information about XSLT and SLAX constructs, see "Summary of XPath and XSLT Constructs" on page 95 and "Summary of SLAX Statements" on page 109.

   For an example that uses both types of instructions and includes a line-by-line analysis of the XSLT syntax, see "Example: Creating Custom Configuration Syntax with Macros" on page 178.

3. Save the script with a meaningful name.

4. Copy the script to either the /var/db/scripts/commit directory on the hard drive or the /config/scripts/commit directory on the flash drive. For information about setting the storage location for commit scripts, see "Storing Commit Scripts in Flash Memory" on page 189.

   If the device has dual Routing Engines and you want the script to take effect on both of them, you must copy the script to the /var/db/scripts/commit or the /config/scripts/commit directory on both Routing Engines. The **commit synchronize** command does not copy scripts between Routing Engines.

5. Enable the script by including the **file** statement at the **[edit system scripts commit]** hierarchy level:

   ```
   [edit system scripts commit]
   file filename;
   ```

   where ***filename*** is the name of the script.

6. If the script makes transient changes, include the **allow-transients** statement at the **[edit system scripts commit]** hierarchy level:

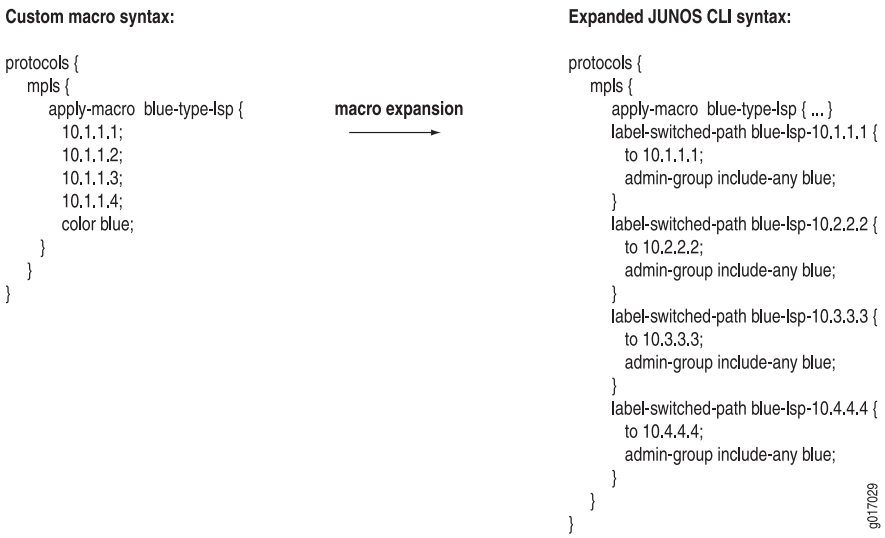
   ```
   [edit system scripts commit]
   allow-transients;
   ```

If all the commit scripts run without errors, any transient changes are loaded into the checkout configuration, but not to the candidate configuration. Any persistent changes are loaded into the candidate configuration. The commit process then continues by validating the configuration and propagating changes to the affected processes on the device running Junos OS.

Example: Creating Custom Configuration Syntax with Macros

Figure 9 on page 179 shows a macro that uses custom syntax and the corresponding expansion to standard Junos OS command-line interface (CLI) syntax.

Figure 9: Sample Macro and Corresponding Junos OS CLI Expansion



In this example, the Junos OS management (mgd) process inspects the configuration, looking for **apply-macro** statements. For each **apply-macro** statement with the **color** parameter included at the **[edit protocols mpls]** hierarchy level, the script generates a transient change, using the data provided within the **apply-macro** statement to expand the macro into a standard Junos OS administrative group for LSPs.

For this example to work, an **apply-macro** statement must be included at the **[edit protocols mpls]** hierarchy level with a set of addresses, a **color**, and a **group-value** parameter. The commit script converts each address to an LSP configuration, and the script converts the **color** parameter into an administrative group.

Following are the commit script instructions that expand the macro in Figure 9 on page 179 and a line-by-line explanation of the script:

XSLT Syntax

```
1  <?xml version="1.0" standalone="yes"?>
2  <xsl:stylesheet version="1.0"
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4     xmlns:junos="http://xml.juniper.net/junos/*/junos"
5     xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
6     xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
7     <xsl:import href="../import/junos.xsl"/>

8     <xsl:template match="configuration">
9        <xsl:variable name="mpls" select="protocols/mpls"/>
10       <xsl:for-each select="$mpls/apply-macro[data/name = 'color']">
11          <xsl:variable name="color" select="data[name = 'color']/value"/>
12          <xsl:for-each select="$mpls/apply-macro[data/name = 'group-value']">
13             <xsl:variable name="group-value" select="data[name = \
                  'group-value']/value"/>
14             <transient-change>
15                <protocols>
16                   <mpls>
17                      <admin-groups>
18                         <name>
19                            <xsl:value-of select="$color"/>
```

```
20                  </name>
21                  <group-value>
22                    <xsl:value-of select="$group-value"/>
23                  </group-value>
24                </admin-groups>
25                <xsl:for-each select="data[not(value)]/name">
26                  <label-switched-path>
27                    <name>
28                      <xsl:value-of select="concat($color, '-lsp-', .)"/>
29                    </name>
30                    <to><xsl:value-of select="."/></to>
31                    <admin-group>
32                      <include-any>
33                        <xsl:value-of select="$color"/>
34                      </include-any>
35                    </admin-group>
36                  </label-switched-path>
37                </xsl:for-each>
38              </mpls>
39            </protocols>
40          </transient-change>
41        </xsl:for-each>
42      </xsl:for-each>
43    </xsl:template>
44  </xsl:stylesheet>
```

Lines 1 through 8 (and Lines 43 and 44) are the boilerplate that you include in every commit script. For brevity, Lines 1 through 8 are omitted here.

Line 9 assigns the **[edit protocols mpls]** hierarchy level to a variable called **$mpls**.

```
9    <xsl:variable name="mpls" select="protocols/mpls"/>
```

Line 10 selects every **apply-macro** statement at the **[edit protocols mpls]** hierarchy level that contains the **color** parameter. The sample configuration in Figure 9 on page 179 contains only one **apply-macro** statement. Therefore, this **<xsl:for-each>** programming instruction takes effect only once.

```
10    <xsl:for-each select="$mpls/apply-macro[data/name = 'color']">
```

Line 11 assigns the value of the **color** parameter, in this case **blue**, to a variable called **$color**.

```
11    <xsl:variable name="color" select="data[name = 'color']/value"/>
```

Line 12 selects every **apply-macro** statement at the **[edit protocols mpls]** hierarchy level that contains the **color** parameter. The sample configuration in Figure 9 on page 179 contains only one **apply-macro** statement. Therefore, this **<xsl:for-each>** programming instruction takes effect only once.

```
12    <xsl:for-each select="$mpls/apply-macro[data/name = 'color']">
```

Line 13 assigns the value of the **group-value** parameter, in this case **0**, to a variable called **$group-value**.

```
13     <xsl:variable name="group-value" select="data[name = 'group-value']/value"/>
```

Lines 14 through 16 generate a transient change at the **[edit protocols mpls]** hierarchy level.

```
14      <transient-change>
15        <protocols>
16          <mpls>
```

Lines 17 through 24 add the **admin-groups** statement to the configuration and assign the value of the **$color** variable to the group name and the value of the **$group-value** variable to the group value.

```
17              <admin-groups>
18                <name>
19                  <xsl:value-of select="$color"/>
20                </name>
21                <group-value>
22                  <<xsl:value-of select="$group-value"/>
23                </group-value>
24              </admin-groups>
```

The resulting configuration statements are as follows:

```
admin-groups {
  blue 0;
}
```

Line 25 selects the name of every parameter that does not already have a value assigned to it, which in this case are the four IP addresses. This **<xsl:for-each>** programming instruction uses recursion through the macro and selects each IP address in turn. The **color** and **group-value** parameters each already have a value assigned (**blue** and **0**, respectively), so this line does not apply to them.

```
25              <xsl:for-each select="data[not(value)]/name">
```

Line 26 adds the **label-switched-path** statement in the configuration.

```
26                <label-switched-path>
```

Lines 27 through 29 assign the **label-switched-path** a name that concatenates the value of the **$color** variable, the text **-lsp-**, and the current IP address currently selected by Line 25 (represented by the " **.**").

```
27                <name>
28                  <xsl:value-of select="concat($color, '-lsp-', .)"/>
29                </name>
```

Line 30 adds the **to** statement to the configuration and sets its value to the IP address currently selected by Line 25.

```
30                <to><xsl:value-of select="."/></to>
```

Lines 31 through 35 add the **admin-group include-any** statement to the configuration and sets its value to the value of the **$color** variable.

```
31              <admin-group>
32                  <include-any>
33                    <xsl:value-of select="$color"/>
34                  </include-any>
35                </admin-group>
```

The resulting configuration statements (for one pass) are as follows:

```
label-switched-path blue-lsp-10.1.1.1 {
  to 10.1.1.1;
  admin-group include-any blue;
}
```

Lines 36 through 42 are closing tags.

```
36                    </label-switched-path>
37                  </xsl:for-each>
38                </mpls>
39              </protocols>
40            </transient-change>
41          </xsl:for-each>
42        </xsl:for-each>
```

Lines 43 and 44 are closing tags for Lines 8 and 2, respectively.

```
43    </xsl:template>
44  </xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  var $mpls = protocols/mpls;
  for-each ($mpls/apply-macro[data/name = 'color']) {
    var $color = data[name = 'color']/value;
    for-each ($mpls/apply-macro[data/name = 'group-value']) {
      var $group-value = data[name='group-value']/value;
      <transient-change> {
        <protocols> {
          <mpls> {
            <admin-groups> {
              <name> $color;
              <group-value> $group-value;
            }
            for-each (data[not(value)]/name) {
              <label-switched-path> {
                <name> $color _ '-lsp-' _ .;
                <to> .;
                <admin-group> {
                  <include-any> $color;
                }
              }
            }
          }
        }
      }
    }
  }
}
```

For more information about this example, see “Example: Configuring Administrative Groups for LSPs” on page 248.

Verifying the Configuration Statements Generated by the Commit Script

To display the configuration statements created by the script, issue the **show protocols mpls | display commit-scripts** command:

```
[edit]
user@host# show protocols mpls | display commit-scripts
apply-macro blue-type-lsp {
   10.1.1.1;
   10.2.2.2;
   10.3.3.3;
   10.4.4.4;
   color blue;
   group-value 0;
}
admin-groups {
   blue 0;
}
label-switched-path blue-lsp-10.1.1.1 {
   to 10.1.1.1;
   admin-group include-any blue;
}
label-switched-path blue-lsp-10.2.2.2 {
   to 10.2.2.2;
   admin-group include-any blue;
}
label-switched-path blue-lsp-10.3.3.3 {
   to 10.3.3.3;
   admin-group include-any blue;
}
label-switched-path blue-lsp-10.4.4.4 {
   to 10.4.4.4;
   admin-group include-any blue;
}
```

CHAPTER 13

Configuring and Troubleshooting Commit Scripts

At commit time, the Junos OS management process (mgd) looks in the /config/scripts/commit or the /var/db/scripts/commit directory, depending on whether the scripts are stored on the flash drive or the hard drive, for one or more commit scripts. Each commit script executes against the candidate configuration database to ensure the configuration conforms to the rules dictated by the scripts.

This chapter discusses the following topics:

Implementing Commit Scripts

To use a commit script on a switch, router, or security device, follow these steps:

1. Write the commit script.

   For information about the types of commit scripts you can write, see “Overview of Generating Custom Warning, Error, and System Log Messages” on page 141, “Overview of Generating Persistent or Transient Configuration Changes” on page 155, and “Overview of Creating Custom Configuration Syntax with Macros” on page 171. For examples, see “Commit Script Examples” on page 201.

2. Copy the script to the /var/db/scripts/commit directory on the hard drive or the /config/scripts/commit directory on the flash drive; for information about setting the storage location for scripts, see “Storing Commit Scripts in Flash Memory” on page 189. Only users who belong to the Junos OS **super-user** login class can access and edit files in these directories.

   NOTE: If the device has dual Routing Engines and you want to enable the commit script to execute on both Routing Engines, you must copy it to the /var/db/scripts/commit or /config/scripts/commit directory on both Routing Engines. The **commit synchronize** command does not automatically copy scripts between Routing Engines.

3. Enable the script by including the **file *filename*** statement at the **[edit system scripts commit]** hierarchy level. For instructions, see “Controlling Execution of Commit Scripts During Commit Operations” on page 186.

4. Issue the **commit** command.

The commit script does not execute during this commit operation, but executes automatically during each subsequent commit operation.

Controlling Execution of Commit Scripts During Commit Operations

Commit scripts are stored on a device’s hard drive in the /var/db/scripts/commit directory or on the flash drive in the /config/scripts/commit directory. Only users in the Junos OS superuser login class can access and edit files in these directories. For information about setting the storage location for scripts, see “Storing Commit Scripts in Flash Memory” on page 189. A commit script is not actually executed during commit operations unless its filename is included at the **[edit system scripts commit file]** hierarchy level. To prevent execution of a commit script, delete the commit script’s filename at that hierarchy level.

By default, the commit operation fails unless all scripts included at the **[edit system scripts commit file]** hierarchy level actually exist in the commit script directory. To enable the commit operation to succeed even if a script is missing, include the **optional** statement at the **[edit system scripts commit file *filename*]** hierarchy level. For example, you might want to mark a script as optional if you anticipate the need to quickly remove it from operation by deleting it from the commit script directory, but do not want to remove the

commit script filename at the **[edit system scripts commit file]** hierarchy level. To enable use of the script again later, you simply replace the file in the commit script directory.

CAUTION: When you include the **optional** statement at the **[edit system scripts commit file *filename*]** hierarchy level, no error message is generated during the commit operation if the file does not exist. As a result, you might not be aware that a script is not executed as you expect.

You can also deactivate and reactivate commit scripts by issuing the **deactivate** and **activate** configuration mode commands. When a commit script is deactivated, the script is marked as inactive in the configuration and does not execute during the commit operation. When a commit script is reactivated, the script is again executed during the commit operation.

To determine which commit scripts are currently active on the device, either list the contents of the /var/run/scripts/commit directory or use the **show** command to display the files included (but not marked **inactive:**) at the **[edit system scripts commit]** hierarchy level.

The filename of a commit script written in SLAX must include the **.slax** extension for the script to be executed. No filename extension is required for commit scripts written in XSLT, but we strongly recommend that you append the **.xsl** extension.

See the following sections:

- Enabling Commit Scripts to Execute During Commit Operations on page 187
- Preventing Commit Scripts from Executing During Commit Operations on page 188
- Deactivating Commit Scripts on page 188
- Activating Commit Scripts on page 188

Enabling Commit Scripts to Execute During Commit Operations

To configure a commit script to execute during a commit operation, follow these steps:

1. Ensure that the commit script is located in the correct directory: the /var/db/scripts/commit directory on the hard drive or the /config/scripts/commit directory on the flash drive. For more information about script storage location, see “Storing Commit Scripts in Flash Memory” on page 189.
2. Enable the commit script by including the **file *filename*** statement at the **[edit system scripts commit]** hierarchy level. Only users who belong to the Junos OS **super-user** login class can enable commit scripts.

   [edit system **scripts commit**]
   user@host# **set file *filename* <optional>**

   - ***filename***—Name of the commit script.
   - **optional**—Enable the commit operation to succeed when the script file does not exist in the script directory. If this statement is omitted, the commit operation fails if the script does not exist.

3. Commit the configuration:

```
[edit]
user@host# commit
```

The commit script does not execute during this commit operation, but executes automatically during each subsequent commit operation.

Preventing Commit Scripts from Executing During Commit Operations

To prevent a commit script from executing during a commit operation, follow these steps:

1. Delete the commit script filename at the **[edit system scripts commit]** hierarchy level:

```
[edit system scripts commit]
user@host# delete file filename
```

filename—Name of the commit script.

2. Remove the commit script from the commit script directory. Although removing the commit script from the commit script directory is not necessary, it is always a good policy to delete unused files from the system.
3. Commit your changes:

```
[edit]
user@host# commit
```

Deactivating Commit Scripts

To deactivate a commit script, follow these steps:

1. Issue the **deactivate** command:

```
[edit]
user@host# deactivate system scripts commit file filename
```

2. Commit your changes:

```
[edit]
user@host# commit
```

A deactivated commit script is marked as **inactive:** and ignored during a commit operation.

In this example, the script mycommit.slax is deactivated:

```
[edit]
user@host# deactivate system scripts commit file mycommit.slax
[edit]
user@host# show system scripts commit
inactive: file mycommit.slax
```

Activating Commit Scripts

To activate an inactive commit script, follow these steps:

1. Issue the **activate** command:

```
[edit]
```

```
user@host# activate system scripts commit file filename
```

2. Commit your changes:

```
[edit]
user@host# commit
```

The commit script does not execute during this commit operation, but executes automatically during each subsequent commit operation.

Storing Commit Scripts in Flash Memory

By default, commit scripts are stored in the /var/db/scripts/commit directory on the device's hard drive. To store them in flash memory instead, include the **load-scripts-from-flash** statement at the **[edit system scripts]** hierarchy level:

```
[edit system scripts]
load-scripts-from-flash;
```

The **load-scripts-from-flash** statement applies to all commit, operation, and event scripts; commit scripts are stored in the /config/scripts/commit directory on the flash drive. Changing the scripts' physical location has no effect on their operation.

NOTE: When you add or remove the load-scripts-from-flash statement in the configuration, you must manually move scripts from the hard drive to the flash drive, or vice versa, as appropriate. They are not moved automatically.

Overview of Updating Commit Scripts from a Remote Source

You can update the commit scripts on a device running Junos OS by retrieving a copy of them from a remote machine (which can be another device running Junos OS or a regular networked computer). This eases file management, because it enables you to update a script in a single location and have devices update their copies from that location. Each device continues to use its locally stored commit scripts during commit operations, only updating a script when you issue the appropriate configuration mode command.

For each commit script, you can define a remote location that houses the master copy of the script, by specifying its URL with the **source** statement at the **[edit system scripts commit file *filename*]** hierarchy level. When you then issue the **set refresh** configuration mode command for a script, the device running Junos OS updates its local copy by retrieving the remote master copy from that URL.

You can also store a copy of a particular script at a remote location other than the master source. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems. When you issue the **set refresh-from** command for the script, you specify the URL for the remote script as an option to the command.

You can use the **set refresh** and **set refresh-from** commands to update either an individual commit script or all enabled commit scripts on the device. When you issue the **set refresh** or **set refresh-from** command, the switch, router, or security device immediately attempts to connect to the appropriate remote source for each script. If successful, the device

updates the local commit script with the remote source. If a problem occurs, a set of error messages is returned.

Issuing the **set refresh** or **set refresh-from** command does not add the **refresh** and **refresh-from** statements to the configuration. In other words, the **set** command behaves differently for these statements than for others: it behaves like an operational mode command by executing an operation, instead of adding a statement to the configuration.

If a device has dual Routing Engines and you want the script to be updated on both Routing Engines, you must include the **refresh** or **refresh-from** statements in the configuration of both Routing Engines. The **commit synchronize** command does not cause the **refresh** or **refresh-from** statement to update scripts on both Routing Engines.

The **refresh** and **refresh-from** statements are mutually exclusive.

CAUTION: We recommend that you do not automate the update function by including the **refresh** statement as a commit script change element. Even though this might seem like a good way to ensure that the most current commit script is always used, we recommend against it for the following reasons:

- Automated update means that the network must be operational for the commit operation to succeed. If the network goes down after you make a configuration error, you cannot recover quickly.
- If multiple commit scripts need to be updated during each commit operation, the network response time can slow down.
- Automated update is always the last action performed during a commit operation. Consequently, the updated commit script executes only during the next commit operation. This is because commit scripts are applied to the candidate configuration before the software copies any persistent changes generated by the scripts to the candidate configuration. For more information, see "Commit Scripts and the Junos OS Commit Model" on page 130. In contrast, if you perform the update operation manually, the updated commit script takes effect as expected, that is, immediately after you commit the **refresh** statement in the configuration.
- If you automate the update operation, the **refresh-from** statement has no effect, because the **refresh-from** URL conflicts with and is overridden by the **source** statement URL. For information about the **refresh-from** statement, see "Updating a Commit Script from an Alternate Location" on page 192.

Related Documentation

- Configuring the Master Source for a Commit Script on page 191
- Updating a Commit Script from the Master Source on page 191
- Updating a Commit Script from an Alternate Location on page 192

Configuring the Master Source for a Commit Script

You can store a master copy of each commit script in a central repository. This eases file management because you can make changes to the master commit script in one place and then update the copy on each device where the commit script is currently enabled.

To specify the location of the master source for a commit script, include the **source** statement at the **[edit system scripts commit file *filename*]** hierarchy level:

```
[edit system scripts commit file filename]
source url;
```

- ***filename***—Name of the commit script.
- ***url***—URL of the commit script's master source file. Specify the source as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

Including the **source** statement in the configuration does not affect the local copy of the commit script until you issue the **set refresh** command as described in "Updating a Commit Script from the Master Source" on page 191. At that point, the master copy is retrieved from the specified URL and overwrites the local copy.

Updating a Commit Script from the Master Source

To update a single commit script from its master source, issue the **set refresh** command at the **[edit system scripts commit file *filename*]** hierarchy level. The master source must already be configured as described in "Configuring the Master Source for a Commit Script" on page 191.

```
[edit system scripts commit file filename]
user@host# set refresh
```

To update all enabled commit scripts from their master sources, issue the **set refresh** command at the **[edit system scripts commit]** hierarchy level:

```
[edit system scripts commit]
user@host# set refresh
```

When you issue the **set refresh** command, the switch, router, or security device immediately attempts to connect to the machine that houses the master source for the script files and retrieve a copy of each file. The master copy overwrites the script stored in the local commit scripts directory. The updated commit script is executed when you next issue the **commit** command. If a master source is not defined for a script, that script is not updated and a warning is issued.

If the device has dual Routing Engines and you want to update a script on both Routing Engines, you must issue the **set refresh** command on each Routing Engine separately. The **commit synchronize** command does not cause the **refresh** statement to update scripts on both Routing Engines.

Updating a Commit Script from an Alternate Location

In addition to updating a commit script from the master source defined by the **source** statement at the **[edit system scripts commit file *filename*]** hierarchy level, you also can update a script from an alternate location. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems. To update a single commit script from the alternate source, issue the **set refresh-from** command at the **[edit system scripts commit file *filename*]** hierarchy level, specifying the location of the remote file:

```
[edit system scripts commit file filename]
user@host# set refresh-from url
```

To update all commit scripts from the alternate source, issue the **set refresh-from** command at the **[edit system scripts commit]** hierarchy level, specifying the location of the remote directory that houses the scripts:

```
[edit system scripts commit]
user@host# set refresh-from url
```

At both hierarchy levels:

url—URL of the remote commit script or directory. Specify the source as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

When you issue the **set refresh-from** command, the switch, router, or security device immediately attempts to connect to the machine that houses the master source for the script files and retrieve a copy of each file. The master copy overwrites the script stored in the local commit scripts directory. The updated commit script is executed when you next issue the **commit** command. If a master source is not defined for a script, that script is not updated and a warning is issued.

If the device has dual Routing Engines and you want to update a script on both Routing Engines, you must issue the **set refresh-from** command on each Routing Engine separately. The **commit synchronize** command does not cause the **refresh-from** statement to update scripts on both Routing Engines.

When you issue the **set refresh-from** command, the Junos OS creates a folder in the /var/tmp directory. This folder is used for file transfer. After the transfer and refresh operations are complete, the Junos OS deletes the temporary folder.

Configuring Checksum Hashes for a Commit Script

You can configure one or more checksum hashes that can be used to verify the integrity of a commit script before the script runs on the switch, router, or security device.

To configure a checksum hash:

1. Create the script.
2. Place the script in the /var/db/scripts/commit directory on the device.

3. Run the script through one or more hash functions to calculate hash values.

   The Junos OS supports MD5, SHA-1, and SHA-256 hash functions.

   ```
   user@host>file checksum md5 /var/db/scripts/commit/script1.slax
     MD5 (/var/db/scripts/commit/script1.slax) = 3af7884eb56e2d4489c2e49b26a39a97
   user@host>file checksum sha1 /var/db/scripts/commit/script1.slax
     SHA1 (/var/db/scripts/commit/script1.slax) =
     00dc690fb08fb049577d012486c9a6dad34212c0
   user@host>file checksum sha-256 /var/db/scripts/commit/script1.slax
     SHA256 (/var/db/scripts/commit/script1.slax) =
     150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
   ```

4. Configure the script.

   ```
   [edit system scripts commit]
   user@host# set file script1.slax checksum md5 3af7884eb56e2d4489c2e49b26a39a97
   [edit system scripts commit]
   user@host# set file script1.slax checksum
     sha-1 00dc690fb08fb049577d012486c9a6dad34212c0
   [edit system scripts commit]
   user@host# set file script1.slax checksum
     sha-256 150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
   ```

   During the execution of the script, the Junos OS recalculates the checksum value using the configured hash and verifies that the calculated value matches the configured value. If the values differ, the execution of the script fails. When you configure multiple checksum values with different hash algorithms, all the configured values must match the calculated values; otherwise, the script execution fails. The commit operation also fails.

Related Documentation

- Configuring Checksum Hashes for an Event Script on page 450
- Configuring Checksum Hashes for an Op Script on page 319
- file checksum md5 command in the *System Basics and Services Command Reference*
- file checksum sha-256 command in the *System Basics and Services Command Reference*
- file checksum sha1 command in the *System Basics and Services Command Reference*

Executing Large Commit Scripts

When you use large commit scripts, the standard commit model can have trouble reading these scripts. When this occurs, you can include the **direct-access** statement at the **[edit system scripts commit]** hierarchy level. When the **direct-access** statement is included, the script driver retrieves the candidate configuration directly from the configuration database. Once the candidate configuration is retrieved, the script driver processes this configuration file against the commit scripts and returns any generated actions to the management (mgd) process.

Directly accessing the configuration data and processing non-XML converted data are processor-intensive compared to the standard commit model. You should only use this feature to handle large files, because system performance is affected.

To set the script driver to directly access the candidate configuration, include the **direct-access** statement at the **[edit system scripts commit]** hierarchy level.

```
[edit system scripts commit]
direct-access;
```

Displaying Commit Script Output

Table 15 on page 194 summarizes the Junos OS command-line interface (CLI) commands you can use to monitor and troubleshoot commit scripts. For more information about the **cscript.log** file, see "Tracing Commit Script Processing" on page 195.

Table 15: Commit Script Configuration and Operational Mode Commands

Task	Command
Configuration Mode Commands	
Display errors and warnings generated by commit scripts.	**commit** or **commit check**
Display detailed information.	**commit | display detail**
Display the underlying Extensible Markup Language (XML) data.	**commit | display xml**
Display the postinheritance contents of the configuration database. This view includes transient changes, but does not include changes made in configuration groups.	**show | display commit-scripts**
Display the postinheritance contents of the configuration database. This view excludes transient changes.	**show | display commit-scripts no-transients**
Display the postinheritance configuration in XML format. Viewing the configuration in XML format can be helpful when you are writing XML Path Language (XPath) expressions and configuration element tags.	**show | display commit-scripts view**
Display the postinheritance configuration in XML format, but exclude transient changes.	**show |** **display commit-scripts view |** **display commit-scripts no-transients**
Display all configuration groups data, including script-generated changes to the groups.	**show groups | display commit-scripts**
Display a particular configuration group, including script-generated changes to the group.	**show groups** *group-name* **| display commit-scripts**

Table 15: Commit Script Configuration and Operational Mode Commands *(continued)*

Task	Command
Operational Mode Commands	
Display logging data associated with all commit script processing.	**show log cscript.log**
Display processing for only the most recent commit operation.	**show log cscript.log | last**
Display processing for script errors.	**show log cscript.log | match error**
Display processing for a particular script.	**show log cscript.log | match *filename***

Tracing Commit Script Processing

Commit script tracing operations track all commit script operations and record them in a log file. The logged error descriptions provide detailed information to help you solve problems faster.

The default operation of commit script tracing is to log important events in a file called cscript.log located in the /var/log directory on the device. When the file cscript.log reaches 128 kilobytes (KB), it is renamed with a number 0 through 9 (in ascending order) appended to the end of the file and then compressed. For example, the log file is saved as cscript.log.0.gz, then cscript.log.1.gz until there are 10 trace files. Then the oldest trace file (cscript.log.9.gz) is overwritten. (For more information about how log files are created, see the *Junos OS System Log Messages Reference*.)

This section discusses the following topics:

Minimum Configuration for Tracing for Commit Script Operations

If no commit script trace options are configured, the simplest way to view the trace output of a commit script is to configure the **output** trace flag and issue the **show log cscript.log | last** command. To do this, perform the following steps:

1. If you have not done so already, enable a commit script by including the **file** statement at the **[edit system scripts commit]** hierarchy level:

   ```
   [edit system scripts commit]
   user@host# set file filename
   ```

2. Enable trace options by including the **traceoptions flag output** statement at the **[edit system scripts commit]** hierarchy level:

   ```
   [edit system scripts commit]
   user@host# set traceoptions flag output
   ```

3. Issue the **commit** command:

```
[edit]
user@host# commit
```

4. Display the resulting trace messages recorded in the file /var/log/cscript.log. At the end of the log is the output generated by the commit script you enabled in Step 1. To display the end of the log, issue the **show log cscript.log | last** operational mode command:

```
[edit]
user@host# run show log cscript.log | last
```

Table 16 on page 196 summarizes useful filtering commands that display selected portions of the **cscript.log** file.

Table 16: Commit Script Tracing Operational Mode Commands

Task	Command
Display logging data associated with all script processing.	**show log cscript.log**
Display script processing for only the most recent commit operation.	**show log cscript.log | last**
Display processing for script errors.	**show log cscript.log | match error**
Display script processing for a particular script.	**show log cscript.log | match *filename***

Example: Minimum Configuration for Enabling Traceoptions for Commit Scripts

Display the trace output for the commit script file **source-route.xsl**:

```
[edit]
system {
  scripts {
    commit {
      file source-route.xsl;
      traceoptions flag output;
    }
  }
}
```

```
[edit]
user@host# commit
[edit]
user@host# run show log cscript.log | last
Jun 20 10:21:24 summary: changes 0, transients 0 (allowed), syslog 0
Jun 20 10:24:15 commit script processing begins
Jun 20 10:24:15 reading commit script configuration
Jun 20 10:24:15 testing commit script configuration
Jun 20 10:24:15 opening commit script '/var/db/scripts/commit/source-route.xsl'
Jun 20 10:24:15 script file '/var/db/scripts/commit/source-route.xsl': size=699;
 md5 = d947972b429d17ce97fe987d94add6fd
Jun 20 10:24:15 reading commit script 'source-route.xsl'
```

```
Jun 20 10:24:15 running commit script 'source-route.xsl'
Jun 20 10:24:15 processing commit script 'source-route.xsl'
Jun 20 10:24:15 results of 'source-route.xsl'
Jun 20 10:24:15 begin dump
<commit-script-output xmlns:junos="http://xml.juniper.net/junos/*/junos"
xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
    <xnm:warning>
        <edit-path>[edit chassis]</edit-path>
        <message>IP source-route processing is not enabled.</message>
    </xnm:warning>
</commit-script-output>Jun 20 10:24:15 end dump
Jun 20 10:24:15 no errors from source-route.xsl
Jun 20 10:24:15 saving commit script changes
Jun 20 10:24:15 summary: changes 0, transients 0 (allowed), syslog 0
```

Configuring Tracing of Commit Scripts

You cannot change the directory (/var/log) to which trace files are written. However, you can customize other trace file settings by including the following statements at the **[edit system scripts commit traceoptions]** hierarchy level:

```
[edit system scripts commit traceoptions]
file <filename> <files number> <size size> <world-readable | no-world-readable>;
flag all;
flag events;
flag input;
flag offline;
flag output;
flag rpc;
flag xslt;
no-remote-trace;
```

These statements are described in the following sections:

- Configuring the Commit Script Log Filename on page 197
- Configuring the Number and Size of Commit Script Log Files on page 197
- Configuring Access to Commit Script Log Files on page 198
- Configuring the Commit Script Trace Operations on page 198

Configuring the Commit Script Log Filename

By default, the name of the file that records trace output is cscript.log. You can specify a different name by including the **file** statement at the **[edit system scripts commit traceoptions]** hierarchy level:

```
[edit system scripts commit traceoptions]
file filename;
```

Configuring the Number and Size of Commit Script Log Files

By default, when the trace file reaches 128 KB in size, it is renamed and compressed to *filename*.0.gz, then *filename*.1.gz, and so on, until there are 10 trace files. Then the oldest trace file (*filename*.9.gz) is overwritten.

You can configure the limits on the number and size of trace files by including the following statements at the **[edit system scripts commit traceoptions file <*filename*>]** hierarchy level:

```
[edit system scripts commit traceoptions file <filename>]
files number size size;
```

For example, set the maximum file size to 640 KB and the maximum number of files to 20. When the file that receives the output of the tracing operation (*filename*) reaches 640 KB, it is renamed and compressed to *filename*.0.gz, and a new file called *filename* is created. When *filename* reaches 640 KB, *filename*.0.gz is renamed *filename*.1.gz and *filename* is renamed and compressed to *filename*.0.gz. This process repeats until there are 20 trace files. Then the oldest file (*filename*.19.gz) is overwritten.

The number of files can range from 2 through 1000 files. The file size can range from 10 KB through 1 gigabyte (GB).

NOTE:

If you set either a maximum file size or a maximum number of trace files, you also must specify the other parameter and a filename.

Configuring Access to Commit Script Log Files

By default, access to the commit script log file is restricted to the owner. You can manually configure access by including the **world-readable** or **no-world-readable** statement at the **[edit system scripts commit traceoptions file <*filename*>]** hierarchy level.

```
[edit system scripts commit traceoptions file <filename>]
(world-readable | no-world-readable);
```

The **no-world-readable** statement restricts commit script log access to the owner. The **world-readable** statement enables unrestricted access to the commit script log file.

Configuring the Commit Script Trace Operations

By default, only important events are logged. You can configure the trace operations to be logged by including the following statements at the **[edit system scripts commit traceoptions]** hierarchy level:

```
[edit system scripts commit traceoptions]
flag all;
flag events;
flag input;
flag offline;
flag output;
flag rpc;
flag xslt;
```

Table 17 on page 199 describes the meaning of the commit script tracing flags.

Table 17: Commit Script Tracing Flags

Flag	Description	Default Setting
all	Trace all operations.	Off
events	Trace important events.	On
input	Trace commit script input data.	Off
offline	Generate data for offline development.	Off
output	Trace commit script output data.	Off
rpc	Trace commit script RPCs.	Off
xslt	Trace the Extensible Stylesheet Language Transformations (XSLT) library.	Off

Troubleshooting Commit Scripts

After you enable a commit script and issue a **commit** command, the commit script takes effect immediately.

Table 18 on page 199 describes some common problems that might occur.

Table 18: Troubleshooting Commit Scripts

Problem	Solution
The output of the **commit check | display detail** command does not reference the expected commit scripts.	Make sure you have enabled all the scripts by including the **file** statement for each one at the **[edit system scripts commit]** hierarchy level.
The output contains the error message: error: could not open commit script: /var/db/scripts/commit/*filename*: No such file or directory	Make sure the file *filename* is in the /var/db/scripts/commit/ directory on your switch, router, or security device.
The following error and warning messages appear: error: invalid transient change generated by commit script: *filename* warning: 1 transient change was generated without [system scripts commit allow-transients]	One of your commit scripts contains instructions to generate a transient change, but you have not enabled transient changes. To rectify this problem, take one of the following actions: • Remove the code that generates a transient change from the indicated script. • Remove the script. • Include the **allow-transients** statement at the **[edit system scripts commit]** hierarchy level.

Table 18: Troubleshooting Commit Scripts *(continued)*

Problem	Solution
An expected action does not occur. For example, a warning message does not appear even though the configuration contains the problem that is supposed to evoke the warning message.	1. Make sure you have enabled the script. Scripts are ignored if they are not enabled. To enable a script, include the **file *filename*** statement at the **[edit system scripts commit]** hierarchy level.
	2. Make sure you have included the required boilerplate in your script. For more information, see "Required Boilerplate for Commit Scripts" on page 133.
	3. Make sure that the Extensible Markup Language Path (XPath) expressions in the script contain valid Junos OS command-line interface (CLI) statements expressed as Junos XML protocol tag elements. You can verify the XML hierarchy by checking the *Junos XML API Configuration Reference* or by issuing the **show configuration | display xml** operational mode command.

	4. Make sure that the programming instructions in the script are referencing the correct context node.
	If you nest one instruction inside another, the outer instruction changes the context node, so the inner instruction must be relative to the outer.
	In the following example, the **<xsl:for-each>** instruction contains an XPath expression, which changes the context node. So the nested **<xsl:if>** instruction uses an XPath expression that is relative to the **interfaces/interface[starts-with(name, 't1-')]** XPath expression.
	`<xsl:for-each select="interfaces/interface[starts-with(name, 't1-')]">`
	`<xsl:if test="not(description)">`

CHAPTER 14

Commit Script Examples

This chapter includes the following examples:

Example: Requiring and Restricting Configuration Statements

This example shows you how to use commit scripts to specify required and prohibited configuration statements.

This commit script ensures that the Ethernet management interface (**fxp0**) is configured and detects when the interface is improperly disabled. The script also detects when the **bgp** statement is not included at the **[edit protocols]** hierarchy level. In all cases, the script emits an error message and the commit operation fails.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:call-template name="error-if-missing">
      <xsl:with-param name="must"
            select="interfaces/interface[name='fxp0']/
            unit[name='0']/family/inet/address"/>
      <xsl:with-param name="statement"
            select="'interfaces fxp0 unit 0 family inet address'"/>
    </xsl:call-template>
    <xsl:call-template name="error-if-present">
      <xsl:with-param name="must"
            select="interfaces/interface[name='fxp0']/disable
            | interfaces/interface[name='fxp0']/
            unit[name='0']/disable"/>
      <xsl:with-param name="message">
        <xsl:text>The fxp0 interface is disabled.</xsl:text>
      </xsl:with-param>
    </xsl:call-template>
    <xsl:call-template name="error-if-missing">
      <xsl:with-param name="must" select="protocols/bgp"/>
      <xsl:with-param name="statement" select="'protocols bgp'"/>
    </xsl:call-template>
  </xsl:template>
  <xsl:template name="error-if-missing">
    <xsl:param name="must"/>
    <xsl:param name="statement" select="'unknown'"/>
    <xsl:param name="message"
            select="'missing mandatory configuration statement'"/>
    <xsl:if test="not($must)">
      <xnm:error>
        <edit-path><xsl:copy-of select="$statement"/></edit-path>
        <message><xsl:copy-of select="$message"/></message>
      </xnm:error>
    </xsl:if>
  </xsl:template>
  <xsl:template name="error-if-present">
    <xsl:param name="must" select="1"/> <!-- give error if param missing -->
    <xsl:param name="message" select="'invalid configuration statement'"/>
    <xsl:for-each select="$must">
      <xnm:error>
        <xsl:call-template name="jcs:edit-path"/>
        <xsl:call-template name="jcs:statement"/>
        <message><xsl:copy-of select="$message"/></message>
      </xnm:error>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  call error-if-missing($must =
        interfaces/interface[name='fxp0']/unit[name='0']/family/inet/address,
        $statement = 'interfaces fxp0 unit 0 family inet address');
  call error-if-present($must = interfaces/interface[name='fxp0']/disable |
        interfaces/interface[name='fxp0']/unit[name='0']/disable) {
    with $message = {
      expr "The fxp0 interface is disabled.";
    }
  }
  call error-if-missing($must = protocols/bgp, $statement = 'protocols bgp');
}
error-if-missing ($must, $statement = 'unknown', $message =
        'missing mandatory configuration statement') {
  if (not($must)) {
    <xnm:error> {
      <edit-path> {
        copy-of $statement;
      }
      <message> {
        copy-of $message;
      }
    }
  }
}
error-if-present ($must = 1, $message = 'invalid configuration statement') {
  for-each ($must) {
    <xnm:error> {
      call jcs:edit-path();
      call jcs:statement();
      <message> {
        copy-of $message;
      }
    }
  }
}
```

Testing the ex-no-nukes Script

To test the ex-no-nukes script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Requiring and Restricting Configuration Statements" on page 201 into a text file, name the file ex-no-nukes.xsl or ex-no-nukes.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-no-nukes.slax**.

```
system {
```

```
    scripts {
        commit {
            file ex-no-nukes.xsl;
        }
    }
}
interfaces {
    fxp0 {
        disable;
        unit 0 {
            family inet {
                address 10.0.0.1/24;
            }
        }
    }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears:

```
[edit]
user@host# commit
[edit interfaces interface fxp0 disable]
  'disable;'
  The fxp0 interface is disabled.
protocols bgp
  missing mandatory configuration statement
error: 2 errors reported by commit scripts
error: commit script failure
```

Example: Requiring Internal Clocking on T1 Interfaces

This example shows you how to use a commit script to require that T1 interfaces be configured with internal clocking.

This commit script ensures that T1 interfaces are explicitly configured to use internal clocking. If the **clocking** statement is not included in the configuration, or if the **clocking external** statement is included, an error message is emitted and the configuration is not committed.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
```

```
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:for-each select="interfaces/interface[starts-with(name, 't1-')]">
      <xsl:variable name="clock-source">
        <xsl:value-of select="clocking"/>
      </xsl:variable>
      <xsl:if test="not($clock-source = 'internal')">
      <!-- or xsl:if test="$clock-source != 'internal'" -->
        <xnm:error>
          <xsl:call-template name="jcs:edit-path"/>
          <xsl:call-template name="jcs:statement">
            <xsl:with-param name="dot" select="clocking"/>
          </xsl:call-template>
          <message>
            This T1 interface should have internal clocking.
          </message>
        </xnm:error>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  for-each (interfaces/interface[starts-with(name, 't1-')]) {
    var $clock-source = {
      expr clocking;
    }
    if (not($clock-source = 'internal')) {
      <xnm:error> {
        call jcs:edit-path();
        call jcs:statement($dot = clocking);
        <message> "This T1 interface should have internal clocking.";
      }
    }
  }
}
```

Testing the ex-clocking-error Script

To test the ex-clocking-error script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Requiring Internal Clocking on T1 Interfaces" on page 204 into a text file, name the file ex-clocking-error.xsl or ex-clocking-error.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.

2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-clocking-error.slax**.

```
system {
    scripts {
        commit {
            file ex-clocking-error.xsl;
        }
    }
}
interfaces {
    t1-0/0/0 {
        clocking external;
    }
    t1-0/0/1 {
        unit 0;
    }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears:

```
[edit]
user@host# commit
[edit interfaces interface t1-0/0/0]
  'clocking external;'
  This T1 interface should have internal clocking.
[edit interfaces interface t1-0/0/1]
  ';'
  This T1 interface should have internal clocking.
error: 2 errors reported by commit scripts
error: commit script failure
```

Example: Imposing a Minimum MTU Setting

The maximum transmission unit (MTU) is the greatest amount of data or packet size (in bytes) that can be transferred in one physical frame on a network.

This example tests the MTU of SONET/SDH interfaces, reports when the MTU is less than the value of the **$min-mtu** variable, here set to 2048, and causes the commit operation to fail.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:param name="min-mtu" select="2048"/>
  <xsl:template match="configuration">
    <xsl:for-each select="interfaces/interface[starts-with(name, 'so-')
                  and mtu and mtu &lt; $min-mtu]">
      <xnm:error>
        <xsl:call-template name="jcs:edit-path"/>
        <xsl:call-template name="jcs:statement">
          <xsl:with-param name="dot" select="mtu"/>
        </xsl:call-template>
        <message>
          <xsl:text>SONET interfaces must have a minimum MTU of </xsl:text>
          <xsl:value-of select="$min-mtu"/>
          <xsl:text>.</xsl:text>
        </message>
      </xnm:error>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

param $min-mtu = 2048;
match configuration {
  for-each (interfaces/interface[starts-with(name, 'so-') and mtu and
            mtu < $min-mtu]) {
    <xnm:error> {
      call jcs:edit-path();
      call jcs:statement($dot = mtu);
      <message> {
        expr "SONET interfaces must have a minimum MTU of ";
        expr $min-mtu;
        expr ".";
      }
```

```
    }
  }
}
```

Testing the ex-so-mtu Script

To test the ex-so-mtu script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Imposing a Minimum MTU Setting" on page 207 into a text file, name the file ex-so-mtu.xsl or ex-so-mtu.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-so-mtu.slax**.

```
system {
  scripts {
    commit {
      file ex-so-mtu.xsl;
    }
  }
}
interfaces {
  so-1/2/2 {
    mtu 2048;
  }
  so-1/2/3 {
    mtu 576;
  }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears:

```
[edit]
user@host# commit
[edit interfaces interface so-1/2/3]
  'mtu 576;'
    SONET interfaces must have a minimum MTU of 2048.
error: 1 error reported by commit scripts
error: commit script failure
```

Example: Limiting the Number of E1 Interfaces

This example limits the number of E1 interfaces configured on a Channelized STM1 Intelligent Queuing (IQ) Physical Interface Card (PIC).

For each channelized STM1 interface (**cstm1-**), the set of corresponding E1 interfaces is selected. The number of those interfaces, as determined by the built-in Extensible Stylesheet Language Transformations (XSLT) **count()** function, cannot exceed the limit set by the global variable **$limit**. If there are more E1 interfaces than **$limit**, a commit error is generated and the commit operation fails.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:param name="limit" select="16"/>
  <xsl:template match="configuration">
    <xsl:variable name="interfaces" select="interfaces"/>
    <xsl:for-each select="$interfaces/interface[starts-with(name, 'cstm1-')]">
      <xsl:variable name="triple" select="substring-after(name, 'cstm1-')"/>
      <xsl:variable name="e1name" select="concat('e1-', $triple)"/>
      <xsl:variable name="count"
          select="count($interfaces/interface[starts-with(name, $e1name)])"/>
      <xsl:if test="$count > $limit">
        <xnm:error>
          <edit-path>[edit interfaces]</edit-path>
          <statement><xsl:value-of select="name"/></statement>
          <message>
            <xsl:text>E1 interface limit exceeded on CSTM1 IQ PIC. </xsl:text>
            <xsl:value-of select="$count"/>
            <xsl:text> E1 interfaces are configured, but only </xsl:text>
            <xsl:value-of select="$limit"/>
            <xsl:text> are allowed.</xsl:text>
          </message>
        </xnm:error>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

param $limit = 16;
match configuration {
  var $interfaces = interfaces;
  for-each ($interfaces/interface[starts-with(name, 'cstm1-')]) {
```

```
    var $triple = substring-after(name, 'cstm1-');
    var $e1name = 'e1-' _ $triple;
    var $count = count($interfaces/interface[starts-with(name, $e1name)]);
    if ($count > $limit) {
      <xnm:error> {
        <edit-path> "[edit interfaces]";
        <statement> name;
        <message> {
          expr "E1 interface limit exceeded on CSTM1 IQ PIC. ";
          expr $count;
          expr " E1 interfaces are configured, but only ";
          expr $limit;
          expr " are allowed.";
        }
      }
    }
  }
}
```

Testing the ex-16-e1-limit Script

To test the ex-16-e1-limit script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Limiting the Number of E1 Interfaces" on page 209 into a text file, name the file ex-16-e1-limit.xsl or ex-16-e1-limit.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-16-e1-limit.slax**.

```
system {
  scripts {
    commit {
      file ex-16-e1-limit.xsl;
    }
  }
}
interfaces {
  cau4-0/1/0 {
    partition 1 interface-type ce1;
    partition 2-18 interface-type e1;
  }
  cstm1-0/1/0 {
    no-partition interface-type cau4;
  }
  ce1-0/1/0:1 {
    clocking internal;
    e1-options {
      framing g704;
    }
    partition 1 timeslots 1-4 interface-type ds;
  }
  ds-0/1/0:1:1 {
    no-keepalives;
    dce;
```

```
    encapsulation frame-relay;
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.0/31;
      }
    }
  }
  e1-0/1/0:2 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
      framing g704;
    }
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.2/31;
      }
    }
  }
  e1-0/1/0:3 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
      framing g704;
    }
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.4/31;
      }
    }
  }
  e1-0/1/0:4 {
    no-keepalives;
    per-unit-scheduler;
    dce;
```

```
    clocking internal;
    encapsulation frame-relay;
    e1-options {
      framing g704;
    }
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.6/31;
      }
    }
  }
  e1-0/1/0:5 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
      framing g704;
    }
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.8/31;
      }
    }
  }
  e1-0/1/0:6 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
      framing g704;
    }
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.10/31;
      }
    }
  }
```

```
e1-0/1/0:7 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
        framing g704;
    }
    lmi {
        lmi-type ansi;
    }
    unit 100 {
        point-to-point;
        dlci 100;
        family inet {
            address 10.0.0.12/31;
        }
    }
}
e1-0/1/0:8 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
        framing g704;
    }
    lmi {
        lmi-type ansi;
    }
    unit 100 {
        point-to-point;
        dlci 100;
        family inet {
            address 10.0.0.14/31;
        }
    }
}
e1-0/1/0:9 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
        framing g704;
    }
    lmi {
        lmi-type ansi;
    }
    unit 100 {
        point-to-point;
        dlci 100;
        family inet {
```

```
            address 10.0.0.16/31;
        }
    }
}
e1-0/1/0:10 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
        framing g704;
    }
    lmi {
        lmi-type ansi;
    }
    unit 100 {
        point-to-point;
        dlci 100;
        family inet {
            address 10.0.0.18/31;
        }
    }
}
e1-0/1/0:11 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
        framing g704;
    }
    lmi {
        lmi-type ansi;
    }
    unit 100 {
        point-to-point;
        dlci 100;
        family inet {
            address 10.0.0.20/31;
        }
    }
}
e1-0/1/0:12 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
        framing g704;
    }
    lmi {
        lmi-type ansi;
    }
```

```
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.22/31;
      }
    }
}
e1-0/1/0:13 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
      framing g704;
    }
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.24/31;
      }
    }
}
e1-0/1/0:14 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
      framing g704;
    }
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.26/31;
      }
    }
}
e1-0/1/0:15 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
      framing g704;
```

```
    }
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.28/31;
      }
    }
  }
  e1-0/1/0:16 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
      framing g704;
    }
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.30/31;
      }
    }
  }
  e1-0/1/0:17 {
    no-keepalives;
    per-unit-scheduler;
    dce;
    clocking internal;
    encapsulation frame-relay;
    e1-options {
      framing g704;
    }
    lmi {
      lmi-type ansi;
    }
    unit 100 {
      point-to-point;
      dlci 100;
      family inet {
        address 10.0.0.32/31;
      }
    }
  }
  e1-0/1/0:18 {
    no-keepalives;
    per-unit-scheduler;
    dce;
```

```
        clocking internal;
        encapsulation frame-relay;
        e1-options {
          framing g704;
        }
        lmi {
          lmi-type ansi;
        }
        unit 100 {
          point-to-point;
          dlci 100;
          family inet {
            address 10.0.0.34/31;
          }
        }
      }
    }
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

   ```
   [edit]
   user@host# load merge terminal
   [Type ^D at a new line to end input]
   ... Paste the contents of the clipboard here ...
   ```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears:

   ```
   [edit]
   user@host# commit
   [edit interfaces]
     'cstm1-0/1/0'
     E1 interface limit exceeded on CSTM1 IQ PIC.
     17 E1 interfaces are configured, but only 16 are allowed.
   error: 1 error reported by commit scripts
   error: commit script failure
   ```

Example: Limiting the Number of ATM Virtual Circuits

This example limits the number of Asynchronous Transfer Mode (ATM) virtual circuits (VCs) configured on an ATM interface.

For each ATM interface, the set of corresponding VCs is selected. The number of those VCs, as determined by the built-in Extensible Stylesheet Language Transformations (XSLT) **count()** function, cannot exceed the limit set by the global variable **$limit**. If there are more ATM VCs than **$limit**, a commit error is generated and the commit operation fails.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
```

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:param name="limit" select="10"/>
  <xsl:template match="configuration">
    <xsl:for-each select="interfaces/interface[starts-with(name, 'at-')]">
      <xsl:variable name="count" select="count(unit)"/>
      <xsl:if test="$count &gt; $limit">
        <xnm:error>
          <edit-path>[edit interfaces]</edit-path>
          <statement><xsl:value-of select="name"/></statement>
          <message>
            <xsl:text>ATM VC limit exceeded; </xsl:text>
            <xsl:value-of select="$count"/>
            <xsl:text> are configured but only </xsl:text>
            <xsl:value-of select="$limit"/>
            <xsl:text> are allowed.</xsl:text>
          </message>
        </xnm:error>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

param $limit = 10;
match configuration {
  for-each (interfaces/interface[starts-with(name, 'at-')]) {
    var $count = count(unit);
    if ($count > $limit) {
      <xnm:error> {
        <edit-path> "[edit interfaces]";
        <statement> name;
        <message> {
          expr "ATM VC limit exceeded; ";
          expr $count;
          expr " are configured but only ";
          expr $limit;
          expr " are allowed.";
        }
      }
    }
  }
}
```

Testing the ex-atm-vc-limit Script

To test the ex-atm-vc-limit script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Limiting the Number of ATM Virtual Circuits" on page 217 into a text file, name the file ex-atm-vc-limit.xsl or ex-atm-vc-limit.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-atm-vc-limit.slax**.

```
system {
    scripts {
        commit {
            file ex-atm-vc-limit.xsl;
        }
    }
}
interfaces {
    at-1/2/3 {
        unit 15 {
            family inet {
                address 10.12.13.15/20;
            }
        }
        unit 16 {
            family inet {
                address 10.12.13.16/20;
            }
        }
        unit 17 {
            family inet {
                address 10.12.13.17/20;
            }
        }
        unit 18 {
            family inet {
                address 10.12.13.18/20;
            }
        }
        unit 19 {
            family inet {
                address 10.12.13.19/20;
            }
        }
        unit 20 {
            family inet {
                address 10.12.13.20/20;
            }
        }
        unit 21 {
            family inet {
                address 10.12.13.21/20;
```

```
        }
      }
      unit 22 {
        family inet {
          address 10.12.13.22/20;
        }
      }
      unit 23 {
        family inet {
          address 10.12.13.23/20;
        }
      }
      unit 24 {
        family inet {
          address 10.12.13.24/20;
        }
      }
      unit 25 {
        family inet {
          address 10.12.13.25/20;
        }
      }
      unit 26 {
        family inet {
          address 10.12.13.26/20;
        }
      }
    }
  }
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears:

```
[edit]
user@host# commit
[edit interfaces]
  'at-1/2/3'
  ATM VC limit exceeded; 12 are configured but only 10 are allowed.
error: 1 error reported by commit scripts
error: commit script failure
```


Example: Controlling IS-IS and MPLS Interfaces

If you want to enable MPLS on an interface, you must make changes at both the **[edit interfaces]** and **[edit protocols mpls]** hierarchy levels. This example shows you how to use commit scripts to decrease the amount of manual configuration.

This example performs two related tasks. If an interface has **[family iso]** configured but not **[family mpls]**, a configuration change is made (using the **<jcs:emit-change>** template) to enable MPLS. MPLS is not valid on loopback interfaces (**lo*X***), so this script ignores loopback interfaces. Secondly, if the interface is not configured at the **[edit protocols mpls]** hierarchy level, a change is made to add the interface. Both changes are accompanied by appropriate warning messages.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:variable name="mpls" select="protocols/mpls"/>
    <xsl:for-each select="interfaces/interface[not(starts-with(name,'lo'))]
                          /unit[family/iso]">
      <xsl:variable name="ifname" select="concat(../name, '.', name)"/>
      <xsl:if test="not(family/mpls)">
        <xsl:call-template name="jcs:emit-change">
          <xsl:with-param name="message">
            <xsl:text>Adding 'family mpls' to ISO-enabled interface</xsl:text>
          </xsl:with-param>
          <xsl:with-param name="content">
            <family>
              <mpls/>
            </family>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:if>
      <xsl:if test="$mpls and not($mpls/interface[name = $ifname])">
        <xsl:call-template name="jcs:emit-change">
          <xsl:with-param name="message">
            <xsl:text>Adding ISO-enabled interface </xsl:text>
            <xsl:value-of select="$ifname"/>
            <xsl:text> to [protocols mpls]</xsl:text>
          </xsl:with-param>
          <xsl:with-param name="dot" select="$mpls"/>
          <xsl:with-param name="content">
            <interface>
              <name>
                <xsl:value-of select="$ifname"/>
              </name>
            </interface>
          </xsl:with-param>
        </xsl:call-template>
```

```
        </xsl:if>
      </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  var $mpls = protocols/mpls;
  for-each (interfaces/interface[not(starts-with(name, "lo"))]/unit[family/iso]) {
    var $ifname = ../name _ '.' _ name;
    if (not(family/mpls)) {
      call jcs:emit-change() {
        with $message = {
          expr "Adding 'family mpls' to ISO-enabled interface";
        }
        with $content = {
          <family> {
            <mpls>;
          }
        }
      }
    }
    if ($mpls and not($mpls/interface[name = $ifname])) {
      call jcs:emit-change($dot = $mpls) {
        with $message = {
          expr "Adding ISO-enabled interface ";
          expr $ifname;
          expr " to [protocols mpls]";
        }
        with $content = {
          <interface> {
            <name> $ifname;
          }
        }
      }
    }
  }
}
```

Testing the ex-iso Script

To test the ex-iso script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Controlling IS-IS and MPLS Interfaces" on page 221 into a text file, name the file ex-iso.xsl or ex-iso.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-iso.slax**.

```
system {
    scripts {
        commit {
            file ex-iso.xsl;
        }
    }
}
interfaces {
    lo0 {
        unit 0 {
            family iso;
        }
    }
    so-1/2/3 {
        unit 0 {
            family iso;
        }
    }
    so-1/3/2 {
        unit 0 {
            family iso;
        }
    }
}
protocols {
    mpls {
        enable;
    }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears:

```
[edit]
user@host# commit
[edit interfaces interface so-1/2/3 unit 0]
  warning: Adding 'family mpls' to ISO-enabled interface
[edit interfaces interface so-1/2/3 unit 0]
  warning: Adding ISO-enabled interface so-1/2/3.0 to [protocols mpls]
[edit interfaces interface so-1/3/2 unit 0]
  warning: Adding 'family mpls' to ISO-enabled interface
[edit interfaces interface so-1/3/2 unit 0]
  warning: Adding ISO-enabled interface so-1/3/2.0 to [protocols mpls]
commit complete
```

5. Issue the **show interfaces** command. Confirm that the loopback interface is not altered, and the SONET/SDH interfaces are altered.

```
[edit]
user@host# show interfaces
so-1/2/3 {
  unit 0 {
    family iso;
    family mpls;
  }
}
so-1/3/2 {
  unit 0 {
    family iso;
    family mpls;
  }
}
lo0 {
  unit 0 {
    family iso;
  }
}
```

Example: Adding T1 Interfaces to a RIP Group

If you want to enable RIP on an interface, you must make changes at both the **[edit interfaces]** and **[edit protocols rip]** hierarchy levels. This example shows you how to use commit scripts to decrease the amount of manual configuration.

This example adds every T1 interface configured at the **[edit interfaces]** hierarchy level to the **[edit protocols rip group test]** hierarchy level. This example includes no error, warning, or system log messages. The changes to the configuration are made silently.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>
```

```
  <xsl:template match="configuration">
    <xsl:variable name="all-t1"
                  select="interfaces/interface[starts-with(name, 't1-')]"/>
    <xsl:if test="$all-t1">
      <change>
        <protocols>
          <rip>
            <group>
              <name>test</name>
              <xsl:for-each select="$all-t1">
                <xsl:variable name="ifname" select="concat(name, '.0')"/>
                <neighbor>
                  <name><xsl:value-of select="$ifname"/></name>
                </neighbor>
              </xsl:for-each>
            </group>
          </rip>
        </protocols>
      </change>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  var $all-t1 = interfaces/interface[starts-with(name, 't1-')];
  if ($all-t1) {
    <change> {
      <protocols> {
        <rip> {
          <group> {
            <name> "test";
            for-each ($all-t1) {
              var $ifname = name _ '.0';
              <neighbor> {
                <name> $ifname;
              }
            }
          }
        }
      }
    }
  }
}
```

Testing the ex-rip-t1 Script

To test the ex-rip-t1 script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Adding T1 Interfaces to a RIP Group" on page 224 into a text file, name the file ex-rip-t1.xsl or ex-rip-t1.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-rip-t1.slax**.

```
system {
    scripts {
        commit {
            file ex-rip-t1.xsl;
        }
    }
}
interfaces {
    t1-0/0/0 {
        unit 0 {
            family iso;
        }
    }
    t1-0/0/1 {
        unit 0 {
            family iso;
        }
    }
    t1-0/0/2 {
        unit 0 {
            family iso;
        }
    }
    t1-0/0/3 {
        unit 0 {
            family iso;
        }
    }
    t1-0/1/0 {
        unit 0 {
            family iso;
        }
    }
    t1-0/1/1 {
        unit 0 {
            family iso;
        }
    }
    t1-0/1/2 {
        unit 0 {
            family iso;
        }
    }
```

```
    t1-0/1/3 {
        unit 0 {
            family iso;
        }
    }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

   ```
   [edit]
   user@host# load merge terminal
   [Type ^D at a new line to end input]
   ... Paste the contents of the clipboard here ...
   ```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command and then the **show protocols rip group test** command. The following output appears:

   ```
   [edit]
   user@host# commit
   user@host# show protocols rip group test
   neighbor t1-0/0/0.0;
   neighbor t1-0/0/1.0;
   neighbor t1-0/0/2.0;
   neighbor t1-0/0/3.0;
   neighbor t1-0/1/0.0;
   neighbor t1-0/1/1.0;
   neighbor t1-0/1/2.0;
   neighbor t1-0/1/3.0;
   ```

Example: Configuring a Default Encapsulation Type

Point-to-Point Protocol (PPP) encapsulation is the default encapsulation type for physical interfaces. You need not configure encapsulation for any physical interfaces that support PPP encapsulation. If you do not configure encapsulation, PPP is used by default. For physical interfaces that do not support PPP encapsulation, you must configure an encapsulation to use for packets transmitted on the interface.

This example configures default Cisco HDLC encapsulation on SONET/SDH interfaces not configured as aggregate interfaces. The **$tag** variable is passed to the **<jcs:emit-change>** template as **transient-change**, so this change is not copied to the candidate configuration.

Simply including configuration groups in the configuration does not enable you to test whether the **aggregate** statement is included for an interface at the **[edit interfaces *interface-name* sonet-options]** hierarchy level. A commit script can perform this test and set the encapsulation only on nonaggregated interfaces. The **ex-so-encap** script written to perform this test has the following syntax:

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:for-each select="interfaces/interface[starts-with(name, 'so-')
                and not(sonet-options/aggregate)]">
      <xsl:call-template name="jcs:emit-change">
        <xsl:with-param name="tag" select="'transient-change'"/>
        <xsl:with-param name="content">
          <encapsulation>cisco-hdlc</encapsulation>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  for-each (interfaces/interface[starts-with(name, 'so-') and
            not(sonet-options/aggregate)]) {
    call jcs:emit-change($tag = 'transient-change') {
      with $content = {
        <encapsulation> "cisco-hdlc";
      }
    }
  }
}
```

Testing the ex-so-encap Script

To test the ex-so-encap script, perform the following steps:

1. Copy the XSLT or SLAX script from “Example: Configuring a Default Encapsulation Type” on page 227 into a text file, name the file ex-so-encap.xsl or ex-so-encap.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-so-encap.slax**.

```
system {
  scripts {
    commit {
      allow-transients;
      file ex-so-encap.xsl;
```

```
        }
      }
    }
    interfaces {
      so-1/2/2 {
        sonet-options {
          aggregate as0;
        }
      }
      so-1/2/3 {
        unit 0 {
          family inet {
            address 10.0.0.3/32;
          }
        }
      }
      so-1/2/4 {
        unit 0 {
          family inet {
            address 10.0.0.4/32;
          }
        }
      }
    }
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

   ```
   [edit]
   user@host# load merge terminal
   [Type ^D at a new line to end input]
   ... Paste the contents of the clipboard here ...
   ```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command.

   ```
   [edit]
   user@host# commit
   ```

When you issue the **commit** command, the ex-so-encap commit script tests for SONET/SDH interfaces that are not configured as aggregate interfaces and sets the default encapsulation type on the nonaggregated interfaces to Cisco HDLC encapsulation. This is implemented as a **transient-change**. Even though the transient changes are in effect, they are not, by default, displayed in the normal output of the **show interfaces** command.

```
[edit]
user@host# show interfaces
so-1/2/2 {
  sonet-options {
    aggregate as0;
```

```
    }
  }
  so-1/2/3 {
    unit 0 {
      family inet {
        address 10.0.0.3/32;
      }
    }
  }
  so-1/2/4 {
    unit 0 {
      family inet {
        address 10.0.0.4/32;
      }
    }
  }
```

To view the configuration with the transient changes, issue the **show interfaces | display commit-scripts** command:

```
[edit]
user@host# show interfaces | display commit-scripts
so-1/2/2 {
  sonet-options {  # The presence of these statements prevents the
    aggregate as0;  # transient change from affecting this interface.
  }
}
so-1/2/3 {
  encapsulation cisco-hdlc;  # Added by transient change.
  unit 0 {
    family inet {
      address 10.0.0.3/32;
    }
  }
}
so-1/2/4 {
  encapsulation cisco-hdlc;  # Added by transient change.
  unit 0 {
    family inet {
      address 10.0.0.4/32;
    }
  }
}
```

Example: Controlling LDP Configuration

If you want to enable LDP on an interface, you must configure the interface at both the **[edit protocols *routing-protocol-name*]** and **[edit protocols ldp]** hierarchy levels. This example shows you how to use commit scripts to ensure that the interface is configured at both levels.

This example tests for interfaces that are configured at either the **[edit protocols ospf]** or **[edit protocols isis]** hierarchy level but not at the **[edit protocols ldp]** hierarchy level. If LDP is not enabled on the device, there is no problem; otherwise, a warning is emitted with the message that the interface does not have LDP enabled.

In case you want some interfaces to be exempt from the LDP test, this script allows you to tag those interfaces as not requiring LDP by including the **apply-macro no-ldp** statement at the **[edit protocols isis interface *interface-name*]** or **[edit protocols ospf area *area-id* interface *interface-name*]** hierarchy level. For example:

```
[edit]
protocols {
  isis {
    interface so-0/1/2.0 {
      apply-macro no-ldp;
    }
  }
}
```

If the **apply-macro no-ldp** statement is included, the warning is not emitted.

A second test ensures that all LDP-enabled interfaces are configured for an interior gateway protocol (IGP). As for LDP, you can exempt some interfaces from the test by including the **apply-macro no-igp** statement at the **[edit protocols ldp interface *interface-name*]** hierarchy level. If that statement is not included and no IGP is configured, a warning is emitted.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:variable name="ldp" select="protocols/ldp"/>
    <xsl:variable name="isis" select="protocols/isis"/>
    <xsl:variable name="ospf" select="protocols/ospf"/>
    <xsl:if test="$ldp">
      <xsl:for-each select="$isis/interface/name |
                            $ospf/area/interface/name">
        <xsl:variable name="ifname" select="."/>
        <xsl:if test="not(../apply-macro[name = 'no-ldp'])
                      and not($ldp/interface[name = $ifname])">
          <xnm:warning>
            <xsl:call-template name="jcs:edit-path"/>
            <xsl:call-template name="jcs:statement"/>
            <message>ldp not enabled for this interface</message>
          </xnm:warning>
        </xsl:if>
      </xsl:for-each>
      <xsl:for-each select="protocols/ldp/interface/name">
        <xsl:variable name="ifname" select="."/>
        <xsl:if test="not(apply-macro[name = 'no-igp'])
                      and not($isis/interface[name = $ifname])
                      and not($ospf/area/interface[name = $ifname])">
          <xnm:warning>
            <xsl:call-template name="jcs:edit-path"/>
            <xsl:call-template name="jcs:statement"/>
```

```
                <message>
                  <xsl:text>ldp-enabled interface does not have </xsl:text>
                  <xsl:text>an IGP configured</xsl:text>
                </message>
              </xnm:warning>
            </xsl:if>
          </xsl:for-each>
        </xsl:if>
      </xsl:template>
    </xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

apply-macro no-ldp;match configuration {
  var $ldp = protocols/ldp;
  var $isis = protocols/isis;
  var $ospf = protocols/ospf;
  if ($ldp) {
    for-each ($isis/interface/name | $ospf/area/interface/name) {
      var $ifname = .;
      if (not(../apply-macro[name = 'no-ldp']) and not($ldp/interface[name =
          $ifname])) {
        <xnm:warning> {
          call jcs:edit-path();
          call jcs:statement();
          <message> "ldp not enabled for this interface";
        }
      }
    }
    for-each (protocols/ldp/interface/name) {
      var $ifname = .;
      if (not(apply-macro[name = 'no-igp']) and not($isis/interface[name =
          $ifname]) and not($ospf/area/interface[name = $ifname])) {
        <xnm:warning> {
          call jcs:edit-path();
          call jcs:statement();
          <message> {
            expr "ldp-enabled interface does not have ";
            expr "an IGP configured";
          }
        }
      }
    }
  }
}
```

Testing the ex-ldp Script

To test the ex-ldp script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Controlling LDP Configuration" on page 230 into a text file, name the file ex-ldp.xsl or ex-ldp.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-ldp.slax**.

```
system {
    scripts {
        commit {
            file ex-ldp.xsl;
        }
    }
}
protocols {
    isis {
        interface so-1/2/2.0 {
            apply-macro no-ldp;
        }
        interface so-1/2/3.0;
    }
    ospf {
        area 10.4.0.0 {
            interface ge-3/2/1.0;
            interface ge-2/2/1.0;
        }
    }
    ldp {
        interface ge-1/2/1.0;
        interface ge-2/2/1.0;
    }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears:

```
[edit]
user@host# commit
```

```
[edit protocols ospf area 10.4.0.0 interface so-1/2/3.0]
    'interface so-1/2/3.0;'
        warning: LDP not enabled for this interface
[edit protocols ospf area 10.4.0.0 interface ge-3/2/1.0]
    'interface ge-3/2/1.0;'
        warning: LDP not enabled for this interface
[edit protocols ldp interface ge-1/2/1.0]
    'interface ge-1/2/1.0;'
        warning: LDP-enabled interface does not have an IGP configured
commit complete
```

Example: Adding a Final then accept Term to a Firewall

Each firewall filter in the Junos OS has an implicit discard action at the end of the filter, which is equivalent to the following explicit filter term:

```
term implicit-rule {
  then discard;
}
```

As a result, if a packet matches none of the terms in the filter, it is discarded. In some cases, you might want to override the default by adding a last term to accept all packets that do not match a firewall filter's series of match conditions. This example adds a final **then accept** action to any firewall filter that does not already end with it.

In this example, the commit script adds a **then accept** statement to any firewall filter that does not already end with an explicit **then accept** statement.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:apply-templates select="firewall/filter | firewall/family/inet
          | firewall/family/inet6" mode="filter"/>
  </xsl:template>
  <xsl:template match="filter" mode="filter">
    <xsl:param name="last" select="term[position() = last()]"/>
    <xsl:comment>
      <xsl:text>Found </xsl:text>
      <xsl:value-of select="name"/>
      <xsl:text>; last </xsl:text>
      <xsl:value-of select="$last/name"/>
    </xsl:comment>
    <xsl:if test="$last and ($last/from or $last/to or not($last/then/accept))">
      <xnm:warning>
        <xsl:call-template name="jcs:edit-path"/>
        <message>
          <xsl:text>filter is missing final 'then accept' rule</xsl:text>
        </message>
      </xnm:warning>
      <xsl:call-template name="jcs:emit-change">
```

```
          <xsl:with-param name="content">
            <term>
              <name>very-last</name>
              <junos:comment>
                <xsl:text>This term was added by a commit script</xsl:text>
              </junos:comment>
              <then>
                <accept/>
              </then>
            </term>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:if>
    </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  apply-templates firewall/filter | firewall/family/inet | firewall/family/inet6 {
    mode "filter";
  }
}
match filter {
  mode "filter";
  param $last = term[position() = last()];
  <xsl:comment> {
    expr "Found ";
    expr name;
    expr "; last ";
    expr $last/name;
  }
  if ($last and ($last/from or $last/to or not($last/then/accept))) {
    <xnm:warning> {
      call jcs:edit-path();
      <message> "filter is missing final 'then accept' rule";
    }
    call jcs:emit-change() {
      with $content = {
        <term> {
          <name> "very-last";
          <junos:comment> "This term was added by a commit script";
          <then> {
            <accept>;
          }
        }
      }
    }
  }
}
```

Testing the ex-add-accept Script

To test the ex-add-accept script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Adding a Final then accept Term to a Firewall" on page 234 into a text file, name the file ex-add-accept.xsl or ex-add-accept.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-add-accept.slax**.

```
system {
    scripts {
        commit {
            file ex-add-accept.xsl;
        }
    }
}
firewall {
    policer sgt-friday {
        if-exceeding {
            bandwidth-percent 10;
            burst-size-limit 250k;
        }
        then discard;
    }
    family inet {
        filter test {
            term one {
                from {
                    interface t1-0/0/0;
                }
                then {
                    count ten-network;
                    discard;
                }
            }
            term two {
                from {
                    forwarding-class assured-forwarding;
                }
                then discard;
            }
        }
    }
}
interfaces {
    t1-0/0/0 {
        unit 0 {
            family inet {
                policer output sgt-friday;
                filter input test;
            }
        }
```

```
    }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears:

```
[edit]
user@host# commit
[edit firewall family inet filter test]
  warning: filter is missing final 'then accept' rule
commit complete
```

5. Issue the **show firewall** command. The following output appears:

```
[edit]
user@host# show firewall
policer sgt-friday {
    if-exceeding {
        bandwidth-percent 10;
        burst-size-limit 250k;
        }
    then discard;
}
family inet {
    filter test {
        term one {
            from {
                interface t1-0/0/0;
            }
            then {
                count ten-network;
                discard;
            }
        }
        term two {
            from {
                forwarding-class assured-forwarding;
            }
            then {
                discard;
            }
        }
        term very-last {
            then accept; /* This term was added by a commit script */
```

```
    }
  }
}
```

Example: Configuring an Interior Gateway Protocol on an Interface

When you add a new interface to an OSPF or IS-IS domain, you must configure the interface at multiple hierarchy levels, including **[edit interfaces]** and **[edit protocols]**. This example uses a macro to automatically include the interface at the **[edit protocols]** hierarchy level and to configure the proper interior gateway protocol (IGP) on the interface, either OSPF or IS-IS, depending on the content of an **apply-macro** statement that you include in the interface configuration. This macro allows you to perform more configuration tasks at a single hierarchy level.

In this example, the Junos OS management (mgd) process inspects the configuration, looking for **apply-macro** statements. For each **apply-macro ifclass** statement included at the **[edit interfaces** ***interface-name*** **unit** ***logical-unit-number*****]** hierarchy level, the script tests whether the **role** parameter is defined as **cpe**. If so, the script checks the **igp** parameter.

If the **igp** parameter is defined as **isis**, the script includes the relevant interface name at the **[edit protocols isis interface]** hierarchy level.

If the **igp** parameter is defined as **ospf**, the script includes the relevant interface name at the **[edit protocols ospf area** ***address*** **interface]** hierarchy level. For OSPF, the script references the **area** parameter to determine the correct subnet address of the area.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:for-each
          select="interfaces/interface/unit/apply-macro[name = 'ifclass']">
      <xsl:variable name="role" select="data[name='role']/value"/>
      <xsl:variable name="igp" select="data[name='igp']/value"/>
      <xsl:variable name="ifname">
        <xsl:value-of select="../../name"/>
        <xsl:text>.</xsl:text>
        <xsl:value-of select="../name"/>
      </xsl:variable>
      <xsl:choose>
        <xsl:when test="$role = 'cpe'">
          <change>
            <xsl:choose>
              <xsl:when test="$igp = 'isis'">
                <protocols>
                  <isis>
                    <interface>
```

```
                        <name><xsl:value-of select="$ifname"/></name>
                      </interface>
                    </isis>
                  </protocols>
                </xsl:when>
                <xsl:when test="$igp = 'ospf'">
                  <protocols>
                    <ospf>
                      <area>
                        <name>
                          <xsl:value-of select="data[name='area']/value"/>
                        </name>
                        <interface>
                          <name><xsl:value-of select="$ifname"/></name>
                        </interface>
                      </area>
                    </ospf>
                  </protocols>
                </xsl:when>
              </xsl:choose>
            </change>
          </xsl:when>
        </xsl:choose>
      </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  for-each (interfaces/interface/unit/apply-macro[name = 'ifclass']) {
    var $role = data[name='role']/value;
    var $igp = data[name='igp']/value;
    var $ifname = {
      expr ../../name;
      expr ".";
      expr ../name;
    }
    if ($role = 'cpe') {
      <change> {
        if ($igp = 'isis') {
          <protocols> {
            <isis> {
              <interface> {
                <name> $ifname;
              }
            }
          }
        }
        else if ($igp = 'ospf') {
          <protocols> {
            <ospf> {
```

```
                <area> {
                  <name> data[name='area']/value;
                  <interface> {
                    <name> $ifname;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
```

Testing the ex-if-class Script

To test the ex-if-class script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Configuring an Interior Gateway Protocol on an Interface" on page 238 into a text file, name the file ex-if-class.xsl or ex-if-class.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-if-class.slax**.

```
system {
  scripts {
    commit {
      file ex-if-class.xsl;
    }
  }
}
interfaces {
  so-1/2/3 {
    unit 0 {
      apply-macro ifclass {
        area 10.4.0.0;
        igp ospf;
        role cpe;
      }
    }
  }
  t3-0/0/0 {
    unit 0 {
      apply-macro ifclass {
        igp isis;
        role cpe;
      }
    }
  }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command.

```
[edit]
user@host# commit
```

Script-Generated Configuration

When you issue the **show protocols** configuration mode command, the following output appears:

```
[edit]
user@host# show protocols
isis {
  interface t3-0/0/0.0;
}
ospf {
  area 10.4.0.0 {
    interface so-1/2/3.0;
  }
}
```

Manual Configuration

When you issue the **show interfaces** configuration mode command, the following output appears:

```
[edit]
user@host# show interfaces
t3-0/0/0 {
  unit 0 {
    apply-macro ifclass {
      igp isis;
      role cpe;
    }
  }
}
so-1/2/3 {
  unit 0 {
    apply-macro ifclass {
      area 10.4.0.0;
      igp ospf;
      role cpe;
    }
  }
}
```

Example: Creating a Complex Configuration Based on a Simple Interface Configuration

This example uses a macro to automatically expand a simple interface configuration by generating a transient change that assigns a default encapsulation type, configures multiple routing protocols on the interface, and applies multiple configuration groups. The Junos OS management (mgd) process inspects the configuration, looking for **apply-macro params** statements included at the **[edit interfaces *interface-name*]** hierarchy level.

When the script finds an **apply-macro params** statement, it performs the following actions:

- Applies the **interface-details** configuration group to the interface.
- Includes the value of the **description** parameter at the **[edit interfaces *interface-name* description]** hierarchy level.
- Includes the value of the **encapsulation** parameter at the **[edit interfaces *interface-name* encapsulation]** hierarchy level. If the **encapsulation** parameter is not included in the **apply-macro params** statement, the script sets the encapsulation to **cisco-hdlc** as a default.
- Sets the logical unit number to **0** and tests whether the **inet-address** parameter is included in the **apply-macro params** statement. If it is, the script includes the value of the **inet-address** parameter at the **[edit interfaces *interface-name* unit 0 family inet address]** hierarchy level.
- Includes the interface name at the **[edit protocols rsvp interface]** hierarchy level.
- Includes the **level 1 enable** and **metric** statements at the **[edit protocols isis interface *interface-name*]** hierarchy level.
- Includes the **level 2 enable** and **metric** statements at the **[edit protocols isis interface *interface-name*]** hierarchy level.
- Tests whether the **isis-level-1** or **isis-level-1-metric** parameter is included in the **apply-macro params** statement. If one or both of these parameters are included, the script includes the **level 1** statement at the **[edit protocols isis interface *interface-name*]** hierarchy level. If the **isis-level-1** parameter is included, the script also includes the value of the **isis-level-1** parameter (**enable** or **disable**) at the **[edit protocols isis interface *interface-name* level 1]** hierarchy level. If the **isis-level-1-metric** parameter is included, the script also includes the value of the **isis-level-1-metric** parameter at the **[edit protocols isis interface *interface-name* level 1 metric]** hierarchy level.
- Tests whether the **isis-level-2** or **isis-level-2-metric** parameter is included in the **apply-macro params** statement. If one or both of these parameters are included, the script includes the **level 2** statement at the **[edit protocols isis interface *interface-name*]** hierarchy level. If the **isis-level-2** parameter is included, the script also includes the value of the **isis-level-2** parameter (**enable** or **disable**) at the **[edit protocols isis interface *interface-name* level 2]** hierarchy level. If the **isis-level-2-metric** parameter is included, the script also includes the value of the **isis-level-2-metric** parameter at the **[edit protocols isis interface *interface-name* level 2 metric]** hierarchy level.
- Includes the interface name at the **[edit protocols ldp interface]** hierarchy level.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:variable name="top" select="."/>
    <xsl:for-each select="interfaces/interface/apply-macro[name = 'params']">
      <xsl:variable name="description"
          select="data[name = 'description']/value"/>
      <xsl:variable name="inet-address"
          select="data[name = 'inet-address']/value"/>
      <xsl:variable name="encapsulation"
          select="data[name = 'encapsulation']/value"/>
      <xsl:variable name="isis-level-1"
          select="data[name = 'isis-level-1']/value"/>
      <xsl:variable name="isis-level-1-metric"
          select="data[name = 'isis-level-1-metric']/value"/>
      <xsl:variable name="isis-level-2"
          select="data[name = 'isis-level-2']/value"/>
      <xsl:variable name="isis-level-2-metric"
          select="data[name = 'isis-level-2-metric']/value"/>
      <xsl:variable name="ifname" select="concat(../name, '.0')"/>
      <transient-change>
        <interfaces>
          <interface>
            <name><xsl:value-of select="../name"/></name>
            <apply-groups>
              <name>interface-details</name>
            </apply-groups>
            <xsl:if test="$description">
              <description>
                <xsl:value-of select="$description"/>
              </description>
            </xsl:if>
            <encapsulation>
              <xsl:choose>
                <xsl:when test="string-length($encapsulation) &gt; 0">
                  <xsl:value-of select="$encapsulation"/>
                </xsl:when>
                <xsl:otherwise>cisco-hdlc</xsl:otherwise>
              </xsl:choose>
            </encapsulation>
            <unit>
              <name>0</name>
              <xsl:if test="string-length($inet-address) &gt; 0">
                <family>
                  <inet>
                    <address>
                      <xsl:value-of select="$inet-address"/>
                    </address>
                  </inet>
                </family>
```

```
            </xsl:if>
          </unit>
        </interface>
      </interfaces>
      <protocols>
        <rsvp>
          <interface>
            <name><xsl:value-of select="$ifname"/></name>
          </interface>
        </rsvp>
        <isis>
          <interface>
            <name><xsl:value-of select="$ifname"/></name>
            <xsl:if test="$isis-level-1 or $isis-level-1-metric">
              <level>
                <name>1</name>
                <xsl:if test="$isis-level-1">
                  <xsl:element name="{$isis-level-1}"/>
                </xsl:if>
                <xsl:if test="$isis-level-1-metric">
                  <metric>
                    <xsl:value-of select="$isis-level-1-metric"/>
                  </metric>
                </xsl:if>
              </level>
            </xsl:if>
            <xsl:if test="$isis-level-2 or $isis-level-2-metric">
              <level>
                <name>2</name>
                <xsl:if test="$isis-level-2">
                  <xsl:element name="{$isis-level-2}"/>
                </xsl:if>
                <xsl:if test="$isis-level-2-metric">
                  <metric>
                    <xsl:value-of select="$isis-level-2-metric"/>
                  </metric>
                </xsl:if>
              </level>
            </xsl:if>
          </interface>
        </isis>
        <ldp>
          <interface>
            <name><xsl:value-of select="$ifname"/></name>
          </interface>
        </ldp>
      </protocols>
    </transient-change>
  </xsl:for-each>
 </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
import "../import/junos.xsl";

match configuration {
  var $top = .;
  for-each (interfaces/interface/apply-macro[name = 'params']) {
    var $description = data[name = 'description']/value;
    var $inet-address = data[name = 'inet-address']/value;
    var $encapsulation = data[name = 'encapsulation']/value;
    var $isis-level-1 = data[name = 'isis-level-1']/value;
    var $isis-level-1-metric = data[name = 'isis-level-1-metric']/value;
    var $isis-level-2 = data[name = 'isis-level-2']/value;
    var $isis-level-2-metric = data[name = 'isis-level-2-metric']/value;
    var $ifname = ../name _ '.0';
    <transient-change> {
      <interfaces> {
        <interface> {
          <name> ../name;
          <apply-groups> {
            <name> "interface-details";
          }
          if ($description) {
            <description> $description;
          }
          <encapsulation> {
            if (string-length($encapsulation) > 0) {
              expr $encapsulation;
            } else {
              expr "cisco-hdlc";
            }
          }
          <unit> {
            <name> "0";
            if (string-length($inet-address) > 0) {
              <family> {
                <inet> {
                  <address> $inet-address;
                }
              }
            }
          }
        }
      }
      <protocols> {
        <rsvp> {
          <interface> {
            <name> $ifname;
          }
        }
        <isis> {
          <interface> {
            <name> $ifname;
            if ($isis-level-1 or $isis-level-1-metric) {
              <level> {
                <name> "1";
                if ($isis-level-1) {
                  <xsl:element name="{$isis-level-1}">;
```

```
                    }
                    if ($isis-level-1-metric) {
                      <metric> $isis-level-1-metric;
                    }
                  }
                }
                if ($isis-level-2 or $isis-level-2-metric) {
                  <level> {
                    <name> "2";
                    if ($isis-level-2) {
                      <xsl:element name="{$isis-level-2}">;
                    }
                    if ($isis-level-2-metric) {
                      <metric> $isis-level-2-metric;
                    }
                  }
                }
              }
            }
            <ldp> {
              <interface> {
                <name> $ifname;
              }
            }
          }
        }
      }
    }
```

Testing the ex-if-params Script

To test the ex-if-params script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Creating a Complex Configuration Based on a Simple Interface Configuration" on page 242 into a text file, name the file ex-if-params.xsl or ex-if-params.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-if-params.slax**.

```
system {
  scripts {
    commit {
      allow-transients;
      file ex-if-params.xsl;
    }
  }
}
groups {
  interface-details {
    interfaces {
      <so-*/*/*> {
        clocking internal;
      }
```

```
        }
      }
    }
    interfaces {
      so-1/2/3 {
        apply-macro params {
          description "Link to Hoverville";
          encapsulation ppp;
          inet-address 10.1.2.3/28;
          isis-level-1 enable;
          isis-level-1-metric 50;
          isis-level-2-metric 85;
        }
      }
    }
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

   ```
   [edit]
   user@host# load merge terminal
   [Type ^D at a new line to end input]
   ... Paste the contents of the clipboard here ...
   ```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command.

   ```
   [edit]
   user@host# commit
   ```

When you issue the **show interfaces | display commit-scripts | display inheritance** configuration mode command, the following output appears:

```
[edit]
user@host# show interfaces | display commit-scripts | display inheritance
so-1/2/3 {
  apply-macro params {
    clocking internal;
    description "Link to Hoverville";
    encapsulation ppp;
    inet-address 10.1.2.3/28;
    isis-level-1 enable;
    isis-level-1-metric 50;
    isis-level-2-metric 85;
  }
  description "Link to Hoverville";
  ##
  ## 'internal' was inherited from group 'interface-details'
  ##
  clocking internal;
  encapsulation ppp;
```

```
    unit 0 {
      family inet {
        address 10.1.2.3/28;
      }
    }
  }
```

When you issue the **show protocols | display commit-scripts** configuration mode command, the following output appears:

```
[edit]
user@host# show protocols | display commit-scripts
rsvp {
  interface so-1/2/3.0;
}
isis {
  interface so-1/2/3.0 {
    level 1 {
      enable;
      metric 50;
    }
    level 2 metric 85;
  }
}
ldp {
  interface so-1/2/3.0;
}
```

Example: Configuring Administrative Groups for LSPs

Administrative groups, also known as link coloring or resource classes, are manually assigned attributes that describe the color of links. Links with the same color conceptually belong to the same class. You can use administrative groups to implement a variety of policy-based label-switched path (LSP) setups.

In this example, the Junos OS management process (mgd) inspects the configuration, looking for **apply-macro** statements. For each **apply-macro** statement with the **color** parameter included at the **[edit protocols mpls]** hierarchy level, the script generates a transient change, using the data provided within the **apply-macro** statement to expand the macro into a standard Junos OS administrative group for LSPs.

For this example to work, an **apply-macro** statement must be included at the **[edit protocols mpls]** hierarchy level with a set of addresses, a **color** parameter, and a **group-value** parameter. The commit script converts each address to an LSP configuration and converts the **color** parameter into an administrative group. For the necessary configuration statements, see “Testing the ex-lsp-admin Script” on page 250.

For a line-by-line explanation of this script, see “Example: Creating Custom Configuration Syntax with Macros” on page 178.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
```

```
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:variable name="mpls" select="protocols/mpls"/>
    <xsl:for-each select="$mpls/apply-macro[data/name = 'color']">
      <xsl:variable name="color" select="data[name = 'color']/value"/>
      <xsl:for-each select="$mpls/apply-macro[data/name = 'group-value']">
        <xsl:variable name="group-value" select="data[name =
              'group-value']/value"/>
        <transient-change>
          <protocols>
            <mpls>
              <admin-groups>
                <name>
                  <xsl:value-of select="$color"/>
                </name>
                <group-value>
                  <xsl:value-of select="$group-value"/>
                </group-value>
              </admin-groups>
              <xsl:for-each select="data[not(value)]/name">
                <label-switched-path>
                  <name>
                    <xsl:value-of select="concat($color, '-lsp-', .)"/>
                  </name>
                  <to><xsl:value-of select="."/></to>
                  <admin-group>
                    <include-any>
                      <xsl:value-of select="$color"/>
                    </include-any>
                  </admin-group>
                </label-switched-path>
              </xsl:for-each>
            </mpls>
          </protocols>
        </transient-change>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  var $mpls = protocols/mpls;
  for-each ($mpls/apply-macro[data/name = 'color']) {
    var $color = data[name = 'color']/value;
    for-each ($mpls/apply-macro[data/name = 'group-value']) {
      var $group-value = data[name = 'group-value']/value;
      <transient-change> {
        <protocols> {
```

```
            <mpls> {
              <admin-groups> {
                <name> $color;
                <group-value> $group-value;
              }
              for-each (data[not(value)]/name) {
                <label-switched-path> {
                  <name> $color _ '-lsp-' _ .;
                  <to> .;
                  <admin-group> {
                    <include-any> $color;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
```

Testing the ex-lsp-admin Script

To test the ex-lsp-admin script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Configuring Administrative Groups for LSPs" on page 248 into a text file, name the file ex-lsp-admin.xsl or ex-lsp-admin.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-lsp-admin.slax**.

```
system {
  scripts {
    commit {
      allow-transients;
      file ex-lsp-admin.xsl;
    }
  }
}
protocols {
  mpls {
    apply-macro blue-type-lsp {
      10.1.1.1;
      10.2.2.2;
      10.3.3.3;
      10.4.4.4;
      color blue;
      group-value 0;
    }
  }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command.

```
[edit]
user@host# commit
```

With Script-Generated Changes

When you issue the **show protocols mpls | display commit-scripts** configuration mode command, the following output appears:

```
[edit]
user@host# show protocols mpls | display commit-scripts
apply-macro blue-type-lsp {
   10.1.1.1;
   10.2.2.2;
   10.3.3.3;
   10.4.4.4;
   color blue;
   group-value 0;
}
admin-groups {
   blue 0;
}
label-switched-path blue-lsp-10.1.1.1 {
   to 10.1.1.1;
   admin-group include-any blue;
}
label-switched-path blue-lsp-10.2.2.2 {
   to 10.2.2.2;
   admin-group include-any blue;
}
label-switched-path blue-lsp-10.3.3.3 {
   to 10.3.3.3;
   admin-group include-any blue;
}
label-switched-path blue-lsp-10.4.4.4 {
   to 10.4.4.4;
   admin-group include-any blue;
}
```

Without Script-Generated Changes

The output of the **show protocols mpls | display commit-scripts no-transients** configuration mode command excludes the **label-switched-path** statements:

```
[edit]
```

```
user@host# show protocols mpls | display commit-scripts no-transients
apply-macro blue-type-lsp {
    10.1.1.1;
    10.2.2.2;
    10.3.3.3;
    10.4.4.4;
    color blue;
    group-value 0;
}
```

When you issue the **show protocols mpls** command without the piped **display commit-scripts no-transients** command, you see the same output because this script does not generate any persistent changes:

```
[edit]
user@host# show protocols mpls
apply-macro blue-type-lsp {
    10.1.1.1;
    10.2.2.2;
    10.3.3.3;
    10.4.4.4;
    color blue;
    group-value 0;
}
```

Example: Configuring Dual Routing Engines

If your device has redundant (also called *dual*) Routing Engines, your Junos OS configuration can be complex. This example shows how you can use commit scripts to simplify and control the configuration of dual Routing Engine platforms.

The Junos OS supports two special configuration groups: **re0** and **re1**. When these groups are applied using the **apply-groups [re0 re1]** statement, they take effect if the Routing Engine name matches the group name. Statements included at the **[edit groups re0]** hierarchy level are inherited only on the Routing Engine named RE0, and statements included at the **[edit groups re1]** hierarchy level are inherited only on the Routing Engine named RE1.

This example includes two commit scripts. The first script, **ex-dual-re.xsl**, emits a warning if the **system host-name** statement, any IP version 4 (IPv4) interface address, or the **fxp0** interface configuration is configured in the target configuration instead of in a configuration group.

The second script, **ex-dual-re2.xsl**, first checks whether the hostname configuration is configured and then checks whether it is configured in a configuration group. The **otherwise** construct emits an error message if the hostname is not configured at all. The first **when** construct allows the script to do nothing if the hostname is already configured in a configuration group. The second **when** construct takes effect when the hostname is configured in the target configuration. In this case, the script generates a transient change that places the hostname configuration into the **re0** and **re1** configuration groups, copies the configured hostname into those groups, concatenates each group hostname with **-RE0** and **-RE1**, and deactivates the hostname in the target configuration so the configuration group hostnames can be inherited.

XSLT Syntax: ex-dual-re.xsl Script

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:for-each select="system/host-name |
                interfaces/interface/unit/family/inet/address |
                interfaces/interface[name = 'fxp0']">
      <xsl:if test="not(@junos:group) or not(starts-with(@junos:group, 're'))">
        <xnm:warning>
          <xsl:call-template name="jcs:edit-path">
            <xsl:with-param name="dot" select=".."/>
          </xsl:call-template>
          <xsl:call-template name="jcs:statement"/>
          <message>
            <xsl:text>statement should not be in target</xsl:text>
            <xsl:text> configuration on dual RE system</xsl:text>
          </message>
        </xnm:warning>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

XSLT Syntax: ex-dual-re2.xsl Script

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:variable name="hn" select="system/host-name"/>
    <xsl:choose>
      <xsl:when test="$hn/@junos:group"/>
      <xsl:when test="$hn">
        <transient-change>
          <groups>
            <name>re0</name>
            <system>
              <host-name>
                <xsl:value-of select="concat($hn, '-RE0')"/>
              </host-name>
            </system>
          </groups>
          <groups>
            <name>re1</name>
            <system>
              <host-name>
                <xsl:value-of select="concat($hn, '-RE1')"/>
```

```
                </host-name>
              </system>
            </groups>
            <system>
              <host-name inactive="inactive"/>
            </system>
          </transient-change>
        </xsl:when>
        <xsl:otherwise>
          <xnm:error>
            <message>Missing [system host-name]</message>
          </xnm:error>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax: ex-dual-re.xsl Script

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  for-each (system/host-name | interfaces/interface/unit/family/inet/address |
                interfaces/interface[name = 'fxp0']) {
    if (not(@junos:group) or not(starts-with(@junos:group, 're'))) {
      <xnm:warning> {
        call jcs:edit-path($dot = ..);
        call jcs:statement();
        <message> {
          expr "statement should not be in target";
          expr " configuration on dual RE system";
        }
      }
    }
  }
}
```

SLAX Syntax: ex-dual-re2.xsl Script

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  var $hn = system/host-name;
  if ($hn/@junos:group) {
  }
  else if ($hn) {
    <transient-change> {
      <groups> {
        <name> "re0";
        <system> {
          <host-name> $hn _ '-RE0';
```

```
        }
      }
      <groups> {
        <name> "re1";
        <system> {
          <host-name> $hn _ '-RE1';
        }
      }
      <system> {
        <host-name inactive="inactive">;
      }
    }
    else {
      <xnm:error> {
        <message> "Missing [system host-name]";
      }
    }
  }
}
```

Testing the ex-dual-re and ex-dual-re2 Scripts

To test the ex-dual-re and ex-dual-re2 scripts, perform the following steps:

1. Copy the XSLT or SLAX scripts from "Example: Configuring Dual Routing Engines" on page 252 into two text files, name the files ex-dual-re.xsl and ex-dual-re2.xsl or ex-dual-re.slax and ex-dual-re2.slax as appropriate, and copy them to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filenames at the **[edit system scripts commit file]** hierarchy level to **ex-dual-re.slax** and **ex-dual-re2.slax**.

```
groups {
  re0 {
    interfaces {
      fxp0 {
        unit 0 {
          family inet {
            address 10.0.0.1/24;
          }
        }
      }
    }
  }
}
apply-groups re0;
system {
  host-name router1;
  scripts {
    commit {
      file ex-dual-re.xsl;
      file ex-dual-re2.xsl;
    }
  }
}
```

```
interfaces {
  fe-0/0/0 {
    unit 0 {
      family inet {
        address 192.168.220.1/30;
      }
    }
  }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

   ```
   [edit]
   user@host# load merge terminal
   [Type ^D at a new line to end input]
   ... Paste the contents of the clipboard here ...
   ```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears. After the commit operation completes, the device hostname is changed to **router1-RE0**.

   ```
   [edit]
   user@host# commit
   [edit system]
     'host-name router1;'
     warning: statement should not be in target configuration on dual RE system
   [edit interfaces interface fe-0/0/0 unit 0 family inet]
     'address 192.168.220.1/30;'
     warning: statement should not be in target configuration on dual RE system
   commit complete
   ```

Example: Preventing Import of the Full Routing Table

In Junos OS routing policy, if you configure a policy with no match conditions and a terminating action of **then accept**, and then apply the policy to a routing protocol, the protocol imports the entire routing table. This example shows how to use a commit script to prevent this scenario.

This example inspects the **import** statements configured at the **[edit protocols ospf]** and **[edit protocols isis]** hierarchy levels to determine if any of the named policies contain a **then accept** term with no match conditions. The script protects against importing the full routing table into these interior gateway protocols (IGPs).

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
```

```
  <xsl:import href="../import/junos.xsl"/>

  <xsl:param name="po"
          select="commit-script-input/configuration/policy-options"/>
  <xsl:template match="configuration">
    <xsl:apply-templates select="protocols/ospf/import"/>
    <xsl:apply-templates select="protocols/isis/import"/>
  </xsl:template>
  <xsl:template match="import">
    <xsl:param name="test" select="."/>
    <xsl:for-each select="$po/policy-statement[name=$test]">
      <xsl:choose>
        <xsl:when test="then/accept and not(to) and not(from)">
          <xnm:error>
            <xsl:call-template name="jcs:edit-path">
              <xsl:with-param name="dot" select="$test"/>
            </xsl:call-template>
            <xsl:call-template name="jcs:statement">
              <xsl:with-param name="dot" select="$test"/>
            </xsl:call-template>
            <message>policy contains bare 'then accept'</message>
          </xnm:error>
        </xsl:when>
      </xsl:choose>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

param $po = commit-script-input/configuration/policy-options;
match configuration {
  apply-templates protocols/ospf/import;
  apply-templates protocols/isis/import;
}
match import {
  param $test = .;
  for-each ($po/policy-statement[name=$test]) {
    if (then/accept and not(to) and not(from)) {
      <xnm:error> {
        call jcs:edit-path($dot = $test);
        call jcs:statement($dot = $test);
        <message> "policy contains bare 'then accept'";
      }
    }
  }
}
```

Testing the ex-import Script

To test the ex-import script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Preventing Import of the Full Routing Table" on page 256 into a text file, name the file ex-import.xsl or ex-import.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-import.slax**.

```
system {
    scripts {
        commit {
            file ex-import.xsl;
        }
    }
}
protocols {
    ospf {
        import bad-news;
    }
}
policy-options {
    policy-statement bad-news {
        then accept;
    }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears:

```
[edit]
user@host# commit
[edit protocols ospf import]
  'import bad-news;'
  policy contains bare 'then accept'
error: 1 error reported by commit scripts
error: commit script failure
```

Example: Automatically Configuring Logical Interfaces and IP Addresses

Every interface you configure requires at least one logical unit and one IP address. Asynchronous Transfer Mode (ATM) interfaces also require a virtual circuit identifier (VCI) for each logical interface. If you need to configure multiple logical units on an interface, you can use a commit script macro to complete the task quickly and with no errors.

This example expands an **apply-macro** statement that provides the name of a physical ATM interface, and a set of parameters that specify how to configure a number of logical units on the interface. The units and VCI numbers are numbered sequentially from the **$unit** variable to the **$max** variable, and are given IP addresses starting at the **$address** variable. To loop through the logical units, Extensible Stylesheet Language Transformations (XSLT) uses recursion, which is implemented in the **<emit-interface>** template. Calculation of the next address is performed in the **<next-address>** template.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:for-each select="interfaces/apply-macro">
      <xsl:variable name="device" select="name"/>
      <xsl:variable name="address" select="data[name='address']/value"/>
      <xsl:variable name="max" select="data[name='max']/value"/>
      <xsl:variable name="unit" select="data[name='unit']/value"/>
      <xsl:variable name="real-max">
        <xsl:choose>
          <xsl:when test="string-length($max) &gt; 0">
            <xsl:value-of select="$max"/>
          </xsl:when>
          <xsl:otherwise>0</xsl:otherwise>
        </xsl:choose>
      </xsl:variable>
      <xsl:variable name="real-unit">
        <xsl:choose>
          <xsl:when test="string-length($unit) &gt; 0">
            <xsl:value-of select="$unit"/>
          </xsl:when>
          <xsl:when test="contains($device, '.')">
            <xsl:value-of select="substring-after($device, '.')"/>
          </xsl:when>
          <xsl:otherwise>0</xsl:otherwise>
        </xsl:choose>
      </xsl:variable>
      <xsl:variable name="real-device">
        <xsl:choose>
          <xsl:when test="contains($device, '.')">
            <xsl:value-of select="substring-before($device, '.')"/>
```

```
        </xsl:when>
        <xsl:otherwise><xsl:value-of select="$device"/></xsl:otherwise>
      </xsl:choose>
    </xsl:variable>
    <transient-change>
      <interfaces>
        <interface>
          <name><xsl:value-of select="$real-device"/></name>
          <xsl:call-template name="emit-interface">
            <xsl:with-param name="address" select="$address"/>
            <xsl:with-param name="unit" select="$real-unit"/>
            <xsl:with-param name="max" select="$real-max"/>
          </xsl:call-template>
        </interface>
      </interfaces>
    </transient-change>
  </xsl:for-each>
</xsl:template>
<xsl:template name="emit-interface">
  <xsl:param name="$max"/>
  <xsl:param name="$unit"/>
  <xsl:param name="$address"/>
  <unit>
    <name><xsl:value-of select="$unit"/></name>
    <vci><xsl:value-of select="$unit"/></vci>
    <family>
      <inet>
        <address><xsl:value-of select="$address"/></address>
      </inet>
    </family>
  </unit>
  <xsl:if test="$max &gt; $unit">
    <xsl:call-template name="emit-interface">
      <xsl:with-param name="address">
        <xsl:call-template name="next-address">
          <xsl:with-param name="address" select="$address"/>
        </xsl:call-template>
      </xsl:with-param>
      <xsl:with-param name="unit" select="$unit + 1"/>
      <xsl:with-param name="max" select="$max"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>
<xsl:template name="next-address">
  <xsl:param name="address"/>
  <xsl:variable name="arg-prefix" select="substring-after($address, '/')"/>
  <xsl:variable name="arg-addr" select="substring-before($address, '/')"/>
  <xsl:variable name="addr">
    <xsl:choose>
      <xsl:when test="string-length($arg-addr) &gt; 0">
        <xsl:value-of select="$arg-addr"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$address"/>
      </xsl:otherwise>
    </xsl:choose>
```

```
    </xsl:variable>
    <xsl:variable name="prefix">
      <xsl:choose>
        <xsl:when test="string-length($arg-prefix) &gt; 0">
          <xsl:value-of select="$arg-prefix"/>
        </xsl:when>
        <xsl:otherwise>32</xsl:otherwise>
      </xsl:choose>
    </xsl:variable>
    <xsl:variable name="a1" select="substring-before($addr, '.')"/>
    <xsl:variable name="a234" select="substring-after($addr, '.')"/>
    <xsl:variable name="a2" select="substring-before($a234, '.')"/>
    <xsl:variable name="a34" select="substring-after($a234, '.')"/>
    <xsl:variable name="a3" select="substring-before($a34, '.')"/>
    <xsl:variable name="a4" select="substring-after($a34, '.')"/>
    <xsl:variable name="r3">
      <xsl:choose>
        <xsl:when test="$a4 &lt; 255">
          <xsl:value-of select="$a3"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="$a3 + 1"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>
    <xsl:variable name="r4">
      <xsl:choose>
        <xsl:when test="$a4 &lt; 255">
          <xsl:value-of select="$a4 + 1"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="0"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>
    <xsl:value-of select="$a1"/>
    <xsl:text>.</xsl:text>
    <xsl:value-of select="$a2"/>
    <xsl:text>.</xsl:text>
    <xsl:value-of select="$r3"/>
    <xsl:text>.</xsl:text>
    <xsl:value-of select="$r4"/>
    <xsl:text>/</xsl:text>
    <xsl:value-of select="$prefix"/>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  for-each (interfaces/apply-macro) {
    var $device = name;
```

```
    var $address = data[name='address']/value;
    var $max = data[name='max']/value;
    var $unit = data[name='unit']/value;
    var $real-max = {
      if (string-length($max) > 0) {
        expr $max;
      } else {
        expr "0";
      }
    }
    var $real-unit = {
      if (string-length($unit) > 0) {
        expr $unit;
      } else if (contains($device, '.')) {
        expr substring-after($device, '.');
      } else {
        expr "0";
      }
    }
    var $real-device = {
      if (contains($device, '.')) {
        expr substring-before($device, '.');
      } else {
        expr $device;
      }
    }
    <transient-change> {
      <interfaces> {
        <interface> {
          <name> $real-device;
          call emit-interface($address, $unit = $real-unit, $max = $real-max);
        }
      }
    }
  }
}
emit-interface ($max, $unit, $address) {
  <unit> {
    <name> $unit;
    <vci> $unit;
    <family> {
      <inet> {
        <address> $address;
      }
    }
  }
  if ($max > $unit) {
    call emit-interface($unit = $unit + 1, $max) {
      with $address = {
        call next-address($address);
      }
    }
  }
}
next-address ($address) {
  var $arg-prefix = substring-after($address, '/');
```

```
  var $arg-addr = substring-before($address, '/');
  var $addr = {
    if (string-length($arg-addr) > 0) {
      expr $arg-addr;
    } else {
      expr $address;
    }
  }
  var $prefix = {
    if (string-length($arg-prefix) > 0) {
      expr $arg-prefix;
    } else {
      expr "32";
    }
  }
  var $a1 = substring-before($addr, '.');
  var $a234 = substring-after($addr, '.');
  var $a2 = substring-before($a234, '.');
  var $a34 = substring-after($a234, '.');
  var $a3 = substring-before($a34, '.');
  var $a4 = substring-after($a34, '.');
  var $r3 = {
    if ($a4 < 255) {
      expr $a3;
    } else {
      expr $a3 + 1;
    }
  }
  var $r4 = {
    if ($a4 < 255) {
      expr $a4 + 1;
    } else {
      expr 0;
    }
  }
  expr $a1;
  expr ".";
  expr $a2;
  expr ".";
  expr $r3;
  expr ".";
  expr $r4;
  expr "/";
  expr $prefix;
}
```

Testing the ex-atm-logical Script

To test the ex-atm-logical script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Automatically Configuring Logical Interfaces and IP Addresses" on page 259 into a text file, name the file ex-atm-logical.xsl or ex-atm-logical.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.

2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-atm-logical.slax**.

```
system {
    scripts {
        commit {
            allow-transients;
            file ex-atm-logical.xsl;
        }
    }
}
interfaces {
    apply-macro at-1/2/3 {
        address 10.12.13.14/20;
        max 200;
        unit 32;
    }
    at-1/2/3 {
        atm-options {
            pic-type atm2;
            vpi 0;
        }
    }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command.

```
[edit]
user@host# commit
```

When you issue the **show interfaces at-1/2/3 | display commit-scripts** configuration mode command, the following output appears:

```
[edit]
user@host# show interfaces at-1/2/3 | display commit-scripts
atm-options {
    pic-type atm2;
    vpi 0;
}
unit 32 {
    vci 32;
    family inet {
        address 10.12.13.14/20;
    }
}
unit 33 {
    vci 33;
    family inet {
        address 10.12.13.15/20;
    }
}
unit 34 {
    vci 34;
    family inet {
        address 10.12.13.16/20;
    }
}
unit 35 {
    vci 35;
    family inet {
        address 10.12.13.17/20;
    }
}
... Logical units 36 through 199 are omitted for brevity ...
unit 200 {
    vci 200 ;
    family inet {
        address 10.12.13.182/20;
    }
}
```

Example: Prepending a Global Policy

For most configuration objects, the order in which the object or its children is created is not significant, because the Junos OS configuration management software stores and displays configuration objects in predetermined positions in the configuration hierarchy. However, some configuration objects—such as routing policies and firewall filters—consist of elements that must be processed and analyzed sequentially in order to produce the intended routing behavior.

This example ensures that a BGP global import policy is applied to all your BGP imports before any other import policies are applied.

This example automatically prepends the **bgp_global_import** policy in front of any other BGP import policies. If the **bgp_global_import** policy statement is not included in the configuration, an error message is emitted, and the commit operation fails.

Otherwise, the commit script uses the **insert="before"** Junos XML protocol attribute and the **position()** XSLT function to control the position of the global BGP policy in relation to any other applied policies. The **insert="before"** attribute inserts the **bgp_global_import** policy in front of the first preexisting BGP import policy.

If there is no preexisting default BGP import policy, the global policy is included in the configuration.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
   <xsl:if       test="not(policy-options/policy-statement[name='bgp_global_import'])">
      <xnm:error>
        <message>Policy error: Policy bgp_global_import required</message>
      </xnm:error>
    </xsl:if>
    <xsl:for-each select="protocols/bgp | protocols/bgp/group |
                          protocols/bgp/group/neighbor">
      <xsl:variable name="first" select="import[position() = 1]"/>
      <xsl:if test="$first">
        <xsl:call-template name="jcs:emit-change">
          <xsl:with-param name="tag" select="'transient-change'"/>
          <xsl:with-param name="content">
            <import insert="before"
                    name="{$first}">bgp_global_import</import>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:if>
    </xsl:for-each>
    <xsl:for-each select="protocols/bgp">
      <xsl:if test="not(import)">
        <xsl:call-template name="jcs:emit-change">
          <xsl:with-param name="tag" select="'transient-change'"/>
          <xsl:with-param name="content">
            <import>bgp_global_import</import>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
```

```
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match configuration {
  if (not(policy-options/policy-statement[name='bgp_global_import'])) {
    <xnm:error> {
      <message> "Policy error: Policy bgp_global_import required";
    }
  }
  for-each (protocols/bgp | protocols/bgp/group
|                         protocols/bgp/group/neighbor) {
    var $first = import[position() = 1];
    if ($first) {
      call jcs:emit-change($tag = 'transient-change') {
        with $content = {
          <import insert="before" name="{$first}"> "bgp_global_import";
        }
      }
    }
  }
  for-each (protocols/bgp) {
    if (not(import)) {
      call jcs:emit-change($tag = 'transient-change') {
        with $content = {
          <import> "bgp_global_import";
        }
      }
    }
  }
}
```

Testing the ex-bgp-global-import Script

To test the ex-bgp-global-import script, perform the following steps:

1. Copy the XSLT or SLAX script from “Example: Prepending a Global Policy” on page 265 into a text file, name the file ex-bgp-global-import.xsl or ex-bgp-global-import.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-bgp-global-import.slax**.

```
system {
  scripts {
    commit {
      allow-transients;
      file ex-bgp-global-import.xsl;
    }
  }
}
interfaces {
  fe-0/0/0 {
    unit 0 {
      family inet {
        address 192.168.16.2/24;
```

```
        }
        family inet6 {
          address 2002:18a5:e996:beef::2/64;
        }
      }
    }
  }
  routing-options {
    autonomous-system 65400;
  }
  protocols {
    bgp {
      group fish {
        neighbor 192.168.16.4 {
          import [ blue green ];
          peer-as 65401;
        }
        neighbor 192.168.16.6 {
          peer-as 65402;
        }
      }
    }
  }
  policy-options {
    policy-statement blue {
      from protocol bgp;
      then accept;
    }
    policy-statement green {
      then accept;
    }
    policy-statement bgp_global_import {
      then accept;
    }
  }
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

   ```
   [edit]
   user@host# load merge terminal
   [Type ^D at a new line to end input]
   ... Paste the contents of the clipboard here ...
   ```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command.

   ```
   [edit]
   user@host# commit
   ```

show protocols

When you issue the **show protocols** configuration mode command, the **bgp_global_import** import policy is not displayed because it is added as a transient change:

```
[edit]
user@host# show protocols
bgp {
  group fish {
    neighbor 192.168.16.4 {
      import [ blue green ];
      peer-as 65401;
    }
    neighbor 192.168.16.6 {
      peer-as 65402;
    }
  }
}
```

show protocols | display commit-scripts

The commit script adds the **import bgp_global_import** statement at the **[edit protocols bgp]** hierarchy level and prepends the **bgp_global_import** policy to the **192.168.16.4** neighbor policy chain:

```
[edit]
user@host# show protocols | display commit-scripts
bgp {
  import bgp_global_import;
  group fish {
    neighbor 192.168.16.4 {
      import [ bgp_global_import blue green ];
      peer-as 65401;
    }
    neighbor 192.168.16.6 {
      peer-as 65402;
    }
  }
}
```

show protocols | display commit-scripts

After you add a policy to the **192.168.16.6** neighbor, which previously had no policies applied, the **bgp_global_import** policy is prepended:

```
[edit]
user@host# set protocols bgp group fish neighbor 192.168.16.6 import green
[edit]
user@host# show protocols | display commit-scripts
bgp {
  import bgp_global_import;
  group fish {
    neighbor 192.168.16.4 {
      import [ bgp_global_import blue green ];
      peer-as 65401;
    }
    neighbor 192.168.16.6 {
      import [ bgp_global_import blue green ];
      peer-as 65402;
    }
  }
}
```

Example: Assigning a Classifier

In Junos OS class of service (CoS), classifiers allow you to associate incoming packets with a forwarding class and loss priority and, based on the associated forwarding class, assign packets to output queues. After you configure a classifier, you must assign it to an input interface.

For each interface configured with the IPv4 protocol family, this script automatically assigns a specified classifier called **fc-q3**.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:template match="configuration">
    <xsl:variable name="cos-all" select="class-of-service"/>
    <xsl:for-each
          select="interfaces/interface[contains(name, '/')]/unit[family/inet]">
      <xsl:variable name="ifname" select="../name"/>
      <xsl:variable name="unit" select="name"/>
      <xsl:variable name="cos"
          select="$cos-all/interfaces[name = $ifname]"/>
      <xsl:if test="not($cos/unit[name = $unit])">
        <xsl:call-template name="jcs:emit-change">
          <xsl:with-param name="message">
            <xsl:text>Adding CoS forwarding class for </xsl:text>
            <xsl:value-of select="concat($ifname, '.', $unit)"/>
          </xsl:with-param>
          <xsl:with-param name="dot" select="$cos-all"/>
          <xsl:with-param name="content">
            <interfaces>
              <name><xsl:value-of select="$ifname"/></name>
              <unit>
                <name><xsl:value-of select="$unit"/></name>
                <forwarding-class>fc-q3</forwarding-class>
              </unit>
            </interfaces>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";
```

```
match configuration {
  var $cos-all = class-of-service;
  for-each (interfaces/interface[contains(name, '/')]/unit[family/inet]) {
    var $ifname = ../name;
    var $unit = name;
    var $cos = $cos-all/interfaces[name = $ifname];
    if (not($cos/unit[name = $unit])) {
      call jcs:emit-change($dot = $cos-all) {
        with $message = {
          expr "Adding CoS forwarding class for ";
          expr $ifname _ '.' _ $unit;
        }
        with $content = {
          <interfaces> {
            <name> $ifname;
            <unit> {
              <name> $unit;
              <forwarding-class> "fc-q3";
            }
          }
        }
      }
    }
  }
}
```

Testing the ex-classifier Script

To test the ex-classifier script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Assigning a Classifier" on page 270 into a text file, name the file ex-classifier.xsl or ex-classifier.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **ex-classifier.slax**.

```
system {
  scripts {
    commit {
      file ex-classifier.xsl;
    }
  }
}
interfaces {
  fe-0/0/0 {
    unit 0 {
      family inet {
        address 10.168.16.2/24;
      }
    }
  }
}
class-of-service {
  forwarding-classes {
```

```
        queue 3 fc-q3;
    }
    classifiers {
        inet-precedence fc-q3 {
            forwarding-class fc-q3 {
                loss-priority low code-points 010;
            }
        }
    }
}
```

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

```
[edit]
user@host# load merge terminal
[Type ^D at a new line to end input]
... Paste the contents of the clipboard here ...
```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command. The following output appears:

```
[edit]
user@host# commit
[edit interfaces interface fe-0/0/0 unit 0]
 warning: Adding CoS forwarding class for fe-0/0/0.0
commit complete
```

The output from the **show class-of-service** configuration mode command now shows that the **fe-0/0/0.0** interface has been assigned the **fc-q3** classifier:

```
[edit]
user@host# show class-of-service
classifiers {
    inet-precedence fc-q3 {
        forwarding-class fc-q3 {
            loss-priority low code-points 010;
        }
    }
}
forwarding-classes {
    queue 3 fc-q3;
}
interfaces {
    fe-0/0/0 {
        unit 0 {
            forwarding-class fc-q3;  # Added by commit script
        }
    }
}
```


Example: Loading a Base Configuration

This script is a macro that sets up a device running Junos OS with a sample base configuration. With minimal manual user input, the script automatically configures:

- A device hostname
- Authentication services
- A superuser login
- System log settings
- Some SNMP settings
- System services, such as FTP and Telnet
- Static routes and a policy to redistribute the static routes
- Configuration groups **re0** and **re1**
- An address for the management Ethernet interface (**fxp0**)
- The loopback interface (**lo0**) with the device ID as the loopback address

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

<xsl:variable name="macro-name" select="'config-system.xsl'"/>
<xsl:template match="configuration">
  <xsl:variable name="rid" select="routing-options/router-id"/>
  <xsl:for-each select="apply-macro[name = 'config-system']">
    <xsl:variable name="hostname" select="data[name =
            'host-name']/value"/>
    <xsl:variable name="fxp0-addr" select="data[name =
            'mgmt-address']/value"/>
    <xsl:variable name="backup-router" select="data[name =
            'backup-router']/value"/>
    <xsl:variable name="bkup-rtr">
      <xsl:choose>
        <xsl:when test="$backup-router">
          <xsl:value-of select="$backup-router"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:variable name="fxp01" select="substring-before($fxp0-addr,
              '.')"/>
          <xsl:variable name="fxp02"
              select="substring-before(substring-after($fxp0-addr, '.'), '.')"/>
          <xsl:variable name="fxp03"
              select="substring-before(substring-after(substring-after(
                  $fxp0-addr, '.'), '.'), '.')"/>
          <xsl:variable name="plen" select="substring-after($fxp0-addr, '/')"/>
```

```
        <xsl:choose>
          <xsl:when test="$plen = 22">
            <xsl:value-of select="concat($fxp01, '.', $fxp02, '.', $fxp03 div
                    4 * 4 + 3, '.254')"/>
          </xsl:when>
          <xsl:when test="$plen = 24">
            <xsl:value-of select="concat($fxp01, '.', $fxp02, '.', $fxp03,
                    '.254')"/>
          </xsl:when>
        </xsl:choose>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsl:choose>
    <xsl:when test="not($rid) or not($hostname) or not($fxp0-addr)">
      <xnm:error>
        <message>
          Must set router ID, host-name and mgmt-address to use this script.
        </message>
      </xnm:error>
    </xsl:when>
    <xsl:otherwise>
      <transient-change>
        <system>
        <!-- Set the following -->
          <domain-name>your-domain.net</domain-name>
          <domain-search>domain.net</domain-search>
          <backup-router>
            <address><xsl:value-of select="$bkup-rtr"/></address>
          </backup-router>
          <time-zone>America/Los_Angeles</time-zone>
          <authentication-order>radius</authentication-order>
          <authentication-order>password</authentication-order>
          <root-authentication>
            <encrypted-password>
              $1$Q3CG88jZ$.qhPUZaHdaIMWF2CvxKTe0
            </encrypted-password>
          </root-authentication>
          <name-server>
            <name>192.168.5.68</name>
          </name-server>
          <name-server>
            <name>172.17.28.100</name>
          </name-server>
          <radius-server>
            <name>192.168.170.241</name>
            <secret>
              $9$4xoDk5T3n/AHkmTQFCA0BIclKWL7sgaRh-bs4GU
            </secret>
          </radius-server>
          <radius-server>
            <name>192.168.4.240</name>
            <secret>
              $9$TQ/t1IcSrKAt0IRheK8X7VYgaZDm5zNdiqmTn6
            </secret>
          </radius-server>
```

```
<login>
  <class>
    <permissions>all</permissions>
  </class>
  <user>
    <name>johnny</name>
    <uid>928</uid>
    <class>superuser</class>
    <authentication>
      <encrypted-password>
        $1$kPU..$w.4FGRAGanJ8U4Yq6sbj7.
      </encrypted-password>
    </authentication>
  </user>
</login>
<services>
  <finger/>
  <ftp/>
  <ssh/>
  <telnet/>
  <xnm-clear-text/>
</services>
<syslog>
  <user>
    <name>*</name>
    <contents>
      <name>any</name>
      <emergency/>
    </contents>
  </user>
  <host>
    <name>host1</name>
    <contents>
      <name>any</name>
      <notice/>
    </contents>
    <contents>
      <name>interactive-commands</name>
      <any/>
    </contents>
  </host>
  <file>
    <name>messages</name>
    <contents>
      <name>any</name>
      <notice/>
    </contents>
    <contents>
      <name>any</name>
      <warning/>
    </contents>
    <contents>
      <name>authorization</name>
      <info/>
    </contents>
    <archive>
```

```
        <world-readable/>
      </archive>
    </file>
    <file>
      <name>security</name>
      <contents>
        <name>interactive-commands</name>
        <any/>
      </contents>
      <archive>
        <world-readable/>
      </archive>
    </file>
  </syslog>
  <processes>
    <routing>
      <undocumented><enable/></undocumented>
    </routing>
    <snmp>
      <undocumented><enable/></undocumented>
    </snmp>
    <ntp>
      <undocumented><enable/></undocumented>
    </ntp>
    <inet-process>
      <undocumented><enable/></undocumented>
    </inet-process>
    <mib-process>
      <undocumented><enable/></undocumented>
    </mib-process>
    <undocumented><management><enable/>
    </undocumented></management>
    <watchdog>
      <enable/>
      </watchdog>
    </processes>
    <ntp>
      <boot-server>domain.net</boot-server>
      <server>
        <name>domainr.net</name>
      </server>
    </ntp>
  </system>
  <snmp>
    <location>Software lab</location>
    <contact>Michael Landon</contact>
    <interface>fxp0.0</interface>
    <community>
      <name>public</name>
      <authorization>read-only</authorization>
      <clients>
        <name>0.0.0.0/0</name>
        <restrict/>
      </clients>
      <clients>
        <name>192.168.1.252/32</name>
```

```
        </clients>
        <clients>
          <name>10.197.169.222/32</name>
        </clients>
        <clients>
          <name>10.197.169.188/32</name>
        </clients>
        <clients>
          <name>10.197.169.193/32</name>
        </clients>
        <clients>
          <name>192.168.65.46/32</name>
        </clients>
        <clients>
          <name>10.209.152.0/23</name>
        </clients>
      </community>
      <community>
        <name>private</name>
        <authorization>read-write</authorization>
        <clients>
          <name>0.0.0.0/0</name>
          <restrict/>
        </clients>
        <clients>
          <name>10.197.169.188/32</name>
        </clients>
      </community>
    </snmp>
    <routing-options>
      <static>
        <junos:comment>/* safety precaution */</junos:comment>
        <route>
          <name>0.0.0.0/0</name>
          <discard/>
          <retain/>
          <no-readvertise/>
        </route>
        <junos:comment>/* corporate net */</junos:comment>
        <route>
          <name>172.16.0.0/12</name>
          <next-hop><xsl:value-of select="$bkup-rtr"/></next-hop>
          <retain/>
          <no-readvertise/>
        </route>
        <junos:comment>/* lab nets */</junos:comment>
        <route>
          <name>192.168.0.0/16</name>
          <next-hop><xsl:value-of select="$bkup-rtr"/></next-hop>
          <retain/>
          <no-readvertise/>
        </route>
        <junos:comment>/* reflector */</junos:comment>
        <route>
          <name>10.17.136.192/32</name>
          <next-hop><xsl:value-of select="$bkup-rtr"/></next-hop>
```

```
        <retain/>
        <no-readvertise/>
      </route>
      <junos:comment>/* another lab1*/</junos:comment>
      <route>
        <name>10.10.0.0/16</name>
        <next-hop><xsl:value-of select="$bkup-rtr"/></next-hop>
        <retain/>
        <no-readvertise/>
      </route>
      <junos:comment>/* ssh servers */</junos:comment>
      <route>
        <name>10.17.136.0/24</name>
        <next-hop><xsl:value-of select="$bkup-rtr"/></next-hop>
        <retain/>
        <no-readvertise/>
      </route>
      <junos:comment>/* Workstations */</junos:comment>
      <route>
        <name>10.150.0.0/16</name>
        <next-hop><xsl:value-of select="$bkup-rtr"/></next-hop>
        <retain/>
        <no-readvertise/>
      </route>
      <junos:comment>/* Hosts */</junos:comment>
      <route>
        <name>10.157.64.0/19</name>
        <next-hop><xsl:value-of select="$bkup-rtr"/></next-hop>
        <retain/>
        <no-readvertise/>
      </route>
      <junos:comment>/* Build Servers */</junos:comment>
      <route>
        <name>10.10.0.0/16</name>
        <next-hop><xsl:value-of select="$bkup-rtr"/></next-hop>
        <retain/>
        <no-readvertise/>
      </route>
    </static>
  </routing-options>
  <policy-options>
    <policy-statement>
      <name>redist</name>
      <from>
        <protocol>static</protocol>
      </from>
      <then>
        <accept/>
      </then>
    </policy-statement>
  </policy-options>
  <apply-groups>re0</apply-groups>
  <apply-groups>re1</apply-groups>
  <groups>
    <name>re0</name>
    <system>
```

```
                    <host-name>
                      <xsl:value-of select="$hostname"/></host-name>
                  </system>
                  <interfaces>
                    <interface>
                      <name>fxp0</name>
                      <unit>
                        <name>0</name>
                        <family>
                          <inet>
                            <address>
                              <name>
                                <xsl:value-of select="$fxp0-addr"/>
                              </name>
                            </address>
                          </inet>
                        </family>
                      </unit>
                    </interface>
                  </interfaces>
                </groups>
                <groups>
                  <name>re1</name>
                </groups>
                <interfaces>
                  <interface>
                    <name>lo0</name>
                    <unit>
                      <name>0</name>
                      <family>
                        <inet>
                          <address>
                            <name><xsl:value-of select="$rid"/></name>
                          </address>
                        </inet>
                      </family>
                    </unit>
                  </interface>
                </interfaces>
              </transient-change>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
      </xsl:template>
    </xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

var $macro-name = 'config-system.xsl';
match configuration {
  var $rid = routing-options/router-id;
  for-each (apply-macro[name = 'config-system']) {
```

```
var $hostname = data[name = 'host-name']/value;
var $fxp0-addr = data[name = 'mgmt-address']/value;
var $backup-router = data[name = 'backup-router']/value;
var $bkup-rtr = {
  if ($backup-router) {
    expr $backup-router;
  }
  else {
    var $fxp01 = substring-before($fxp0-addr,                    '.');
    var $fxp02 = substring-before(substring-after($fxp0-addr, '.'), '.');
    var $fxp03 = substring-before(substring-after(substring-after(
                $fxp0- addr, '.'), '.'), '.');
    var $plen = substring-after($fxp0-addr, '/');
    if ($plen = 22) {
      expr $fxp01 _ '.' _ $fxp02 _ '.' _ $fxp03 div 4 * 4 + 3 _ '.254';
    }
    else if ($plen = 24) {
      expr $fxp01 _ '.' _ $fxp02 _ '.' _ $fxp03 _ '.254';
    }
  }
}
if (not($rid) or not($hostname) or not($fxp0-addr)) {
  <xnm:error> {
    <message> "Must set router ID, host-name, and mgmt-address to use
              this script.";
  }
}
else {
  <transient-change> {
    <system> {
      /* Set the following */
      <domain-name> "your-domain.net";
      <domain-search> "domain.net";
      <backup-router> {
        <address> $bkup-rtr;
      }
      <time-zone> "America/Los_Angeles";
      <authentication-order> "radius";
      <authentication-order> "password";
      <root-authentication> {
        <encrypted-password>
              "$1$Q3CG88jZ$.qhPUZaHdaIMWF2CvxKTe0";
      }
      <name-server> {
        <name> "192.168.5.68";
      }
      <name-server> {
        <name> "172.17.28.100";
      }
      <radius-server> {
        <name> "192.168.170.241";
        <secret> "$9$4xoDk5T3n/AHkmTQFCA0BIclKWL7sgaRh-bs4GU";
      }
      <radius-server> {
        <name> "192.168.4.240";
        <secret> "$9$TQ/t1IcSrKAt0IRheK8X7VYgaZDm5zNdiqmTn6";
```

```
    }
    <login> {
      <class> {
        <permissions> "all";
      }
      <user> {
        <name> "johnny";
        <uid> "928";
        <class> "superuser";
        <authentication> {
          <encrypted-password>"$1$kPU..$w.4FGRAGanJ8U4Yq6sbj7.";
      }
    }
  }
  <services> {
    <finger>;
    <ftp>;
    <ssh>;
    <telnet>;
    <xnm-clear-text>;
  }
  <syslog> {
    <user> {
      <name> "*";
      <contents> {
        <name> "any";
        <emergency>;
      }
    }
    <host> {
      <name> "host1";
      <contents> {
        <name> "any";
        <notice>;
      }
      <contents> {
        <name> "interactive-commands";
        <any>;
      }
    }
    <file> {
      <name> "messages";
      <contents> {
        <name> "any";
        <notice>;
      }
      <contents> {
        <name> "any";
        <warning>;
      }
      <contents> {
        <name> "authorization";
        <info>;
      }
      <archive> {
        <world-readable>;
```

```
    }
  }
  <file> {
    <name> "security";
    <contents> {
      <name> "interactive-commands";
      <any>;
    }
    <archive> {
      <world-readable>;
    }
  }
}
<processes> {
  <routing> {
    <undocumented><enable>;
  }
  <snmp> {
    <undocumented><enable>;
  }
  <ntp> {
    <undocumented><enable>;
  }
  <inet-process> {
    <undocumented> <enable>;
  }
  <mib-process> {
    <undocumented> <enable>;
  }
  <undocumented><management> {
    <enable>;
  }
  <watchdog> {
    <enable>;
  }
  <ntp> {
    <boot-server> "domain.net";
    <server> {
      <name> "domainr.net";
    }
  }
}
<snmp> {
  <location> "Software lab";
  <contact> "Michael Landon";
  <interface> "fxp0.0";
  <community> {
    <name> "public";
    <authorization> "read-only";
    <clients> {
      <name> "0.0.0.0/0";
      <restrict>;
    }
    <clients> {
      <name> "192.168.1.252/32";
    }
```

```
        <clients> {
          <name> "10.197.169.222/32";
        }
        <clients> {
          <name> "10.197.169.188/32";
        }
        <clients> {
          <name> "10.197.169.193/32";
        }
        <clients> {
          <name> "192.168.65.46/32";
        }
        <clients> {
          <name> "10.209.152.0/23";
        }
      }
      <community> {
        <name> "private";
        <authorization> "read-write";
        <clients> {
          <name> "0.0.0.0/0";
          <restrict>;
        }
        <clients> {
          <name> "10.197.169.188/32";
        }
      }
    }
    <routing-options> {
      <static> {
        <junos:comment> "/* safety precaution */";
        <route> {
          <name> "0.0.0.0/0";
          <discard>;
          <retain>;
          <no-readvertise>;
        }
        <junos:comment> "/* corporate net */";
        <route> {
          <name> "172.16.0.0/12";
          <next-hop> $bkup-rtr;
          <retain>;
          <no-readvertise>;
        }
        <junos:comment> "/* lab nets */";
        <route> {
          <name> "192.168.0.0/16";
          <next-hop> $bkup-rtr;
          <retain>;
          <no-readvertise>;
        }
        <junos:comment> "/* reflector */";
        <route> {
          <name> "10.17.136.192/32";
          <next-hop> $bkup-rtr;
          <retain>;
```

```
        <no-readvertise>;
      }
      <junos:comment> "/* another lab1*/";
      <route> {
        <name> "10.10.0.0/16";
        <next-hop> $bkup-rtr;
        <retain>;
        <no-readvertise>;
      }
      <junos:comment> "/* ssh servers */";
      <route> {
        <name> "10.17.136.0/24";
        <next-hop> $bkup-rtr;
        <retain>;
        <no-readvertise>;
      }
      <junos:comment> "/* Workstations */";
      <route> {
        <name> "10.150.0.0/16";
        <next-hop> $bkup-rtr;
        <retain>;
        <no-readvertise>;
      }
      <junos:comment> "/* Hosts */";
      <route> {
        <name> "10.157.64.0/19";
        <next-hop> $bkup-rtr;
        <retain>;
        <no-readvertise>;
      }
      <junos:comment> "/* Build Servers */";
      <route> {
        <name> "10.10.0.0/16";
        <next-hop> $bkup-rtr;
        <retain>;
        <no-readvertise>;
      }
    }
  }
  <policy-options> {
    <policy-statement> {
      <name> "redist";
      <from> {
        <protocol> "static";
      }
      <then> {
        <accept>;
      }
    }
  }
  <apply-groups> "re0";
  <apply-groups> "re1";
  <groups> {
    <name> "re0";
    <system> {
      <host-name> $hostname;
```

```
        }
        <interfaces> {
          <interface> {
            <name> "fxp0";
            <unit> {
              <name> "0";
              <family> {
                <inet> {
                  <address> {
                    <name> $fxp0-addr;
                  }
                }
              }
            }
          }
        }
        <groups> {
          <name> "re1";
        }
        <interfaces> {
          <interface> {
            <name> "lo0";
            <unit> {
              <name> "0";
              <family> {
                <inet> {
                  <address> {
                    <name> $rid;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Testing the config-system Script

To test the config-system script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Loading a Base Configuration" on page 273 into a text file, name the file config-system.xsl or config-system.slax as appropriate, and copy it to the /var/db/scripts/commit directory on the device.
2. Select the following configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX version of the script, change the filename at the **[edit system scripts commit file]** hierarchy level to **config-system.slax**.

```
system {
  scripts {
    commit {
      allow-transients;
```

```
            file config-system.xsl;
        }
      }
    }
    apply-macro config-system {
      host-name test;
      mgmt-address 10.0.0.1/32;
      backup-router 10.0.0.2;
    }
```

The **host-name** and **mgmt-address** statements are mandatory. The **backup-router** statement is optional. You can substitute a hostname, a management Ethernet (**fxp0**) IP address, and a backup router IP address that are appropriate to your device.

3. In configuration mode, issue the **load merge terminal** command to merge the stanzas into your device configuration:

   ```
   [edit]
   user@host# load merge terminal
   [Type ^D at a new line to end input]
   ... Paste the contents of the clipboard here ...
   ```

   a. At the prompt, paste the contents of the clipboard using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the **commit** command.

   ```
   [edit]
   user@host# commit
   ```

After committing the configuration, issue the **show | display commit-scripts** configuration mode command to view the device base configuration:

```
user@host# show | display commit-scripts
...
```

CHAPTER 15

Summary of Junos XML and XSLT Tag Elements Used in Commit Scripts

This chapter lists the Junos XML tag elements used to generate custom warning, error, and system log (syslog) messages, and the XSLT tag elements used to make custom permanent or transient changes. The tag elements are in alphabetical order.

<change> (XSLT)

Usage

```
<change>
  <!-- tag elements representing configuration statements to load -->
</change>
```

Release Information Statement introduced in Junos OS Release 7.4.

Description Request that the Junos XML protocol server load configuration data into the candidate configuration by enclosing the configuration data within an opening **<change>** tag and closing **</change>** tag. Inside the **<change>** element, include the configuration data as Junos XML tag elements.

Usage Guidelines See "Overview of Generating Persistent or Transient Configuration Changes" on page 155 and "Overview of Creating Custom Configuration Syntax with Macros" on page 171.

Related Documentation

- <transient-change> (XSLT) on page 288

<syslog> (Junos XML)

Usage

```
<syslog="namespace-URL" xmlns:xnm="namespace-URL">
  <message>syslog-message </message>
</syslog>
```

Release Information Statement introduced in Junos OS Release 7.4.

Description Record events that occur on a device running Junos OS.

Attributes **xmlns**—Names the Extensible Markup Language (XML) namespace for the contents of the tag element. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

xmlns:xnm—Names the XML namespace for child tag elements that have the **xnm:** prefix on their names. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

Contents **<message>**—Specifies the content of the system log message in a natural-language text string.

Usage Guidelines See "Generating a Custom Warning, Error, or System Log Message" on page 142.

<transient-change> (XSLT)

Usage

```
<transient-change>
  <!-- tag elements representing configuration statements to load -->
</transient-change>
```

Release Information Statement introduced in Junos OS Release 7.4.

Description Request that the Junos XML protocol server load configuration data into the checkout configuration by enclosing the configuration data within an opening **<transient-change>** and closing **</transient-change>** tag. Inside the **<transient-change>** element, include the configuration data as Junos XML tag elements.

Usage Guidelines See "Overview of Generating Persistent or Transient Configuration Changes" on page 155 and "Overview of Creating Custom Configuration Syntax with Macros" on page 171.

Related Documentation

- <change> (XSLT) on page 287

xnm:error (Junos XML)

Usage

```
<xnm:error xmlns="namespace-URL" xmlns:xnm="namespace-URL">
  <parse/>
  <source-daemon>module-name</source-daemon>
  <filename>filename</filename>
  <line-number>line-number</line-number>
  <column>column-number</column>
  <token>input-token-id</token>
  <edit-path>edit-path-name</edit-path>
  <statement>statement-string</statement>
  <message>error-string</message>
  <re-name>re-name-string</re-name>
  <database-status-information>user</database-status-information>
  <reason>reason-string</reason>
</xnm:error>
```

Release Information Statement introduced in Junos OS Release 7.4.

Description Indicate that the commit script has detected an error in the configuration and has caused the commit operation to fail. The child tag elements described in the Contents section detail the nature of the error.

Attributes **xmlns**—Names the XML namespace for the contents of the tag element. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

xmlns:xnm—Names the XML namespace for child tag elements that have the **xnm:** prefix on their names. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

Contents **<column>**—Identifies the element that caused the error by specifying its position as the number of characters after the first character in the line specified by the **<line-number>** tag element in the configuration file that was being loaded (which is named in the **<filename>** tag element).

<database-status-information>—Provides information about the users currently editing the configuration.

<edit-path>—Specifies the command-line interface (CLI) configuration mode edit path in effect when the error occurred (provided only during loading of a configuration file).

<filename>—Names the configuration file that was being loaded.

<line-number>—Specifies the line number where the error occurred in the configuration file that was being loaded, which is named by the **<filename>** tag element.

<message>—Describes the error in a natural-language text string.

<parse/>—Indicates that there was a syntactic error in the request submitted by the client application.

<re-name>—Names the Routing Engine on which the <source-daemon> is running.

<reason>—Describes the reason for the error.

<source-daemon>—Names the Junos OS module that was processing the request in which the error occurred.

<statement>—Specifies the configuration statement in effect when the problem occurred.

<token>—Names the element in the request that caused the error.

Usage Guidelines See "Generating a Custom Warning, Error, or System Log Message" on page 142.

Related Documentation

- xnm:warning (Junos XML) on page 290

xnm:warning (Junos XML)

Usage

```
<xnm:warning xmlns="namespace-URL" xmlns:xnm="namespace-URL">
  <source-daemon>module-name</source-daemon>
  <filename>filename</filename>
  <line-number>line-number</line-number>
  <column>column-number</column>
  <token>input-token-id</token>
  <edit-path>edit-path-name</edit-path>
  <statement>statement-name</statement>
  <message>error-string</message>
  <reason>reason-string</reason>
</xnm:warning>
```

Release Information Statement introduced in Junos OS Release 7.4.

Description Indicate that the commit script has encountered a problem with the configuration. The child tag elements described in the Contents section detail the nature of the warning.

Attributes **xmlns**—Names the XML namespace for the contents of the tag element. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

xmlns:xnm—Names the XML namespace for child tag elements that have the **xnm:** prefix on their names. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as **1.1**.

Contents **<column>**—Identifies the element that caused the warning by specifying its position as the number of characters after the first character in the line specified by the **<line-number>** tag element in the configuration file that was being loaded (which is named in the **<filename>** tag element).

<edit-path>—Specifies the CLI configuration mode edit path in effect when the problem occurred (provided only during loading of a configuration file).

<filename>—Names the configuration file that was being loaded.

<line-number>—Specifies the line number where the problem occurred in the configuration file that was being loaded, which is named by the **<filename>** tag element.

<message>—Describes the warning in a natural-language text string.

<reason>—Describes the reason for the warning.

<source-daemon>—Names the Junos OS module that was processing the request in which the problem occurred.

<statement>—Names the configuration statement in effect when the problem occurred.

<token>—Names which element in the request caused the warning.

Usage Guidelines See "Generating a Custom Warning, Error, or System Log Message" on page 142

Related Documentation

- xnm:error (Junos XML) on page 289.

CHAPTER 16

Summary of Commit Script Configuration Statements

This chapter describes each configuration statement for commit scripts. The statements are organized alphabetically.

allow-transients

Syntax allow-transients;

Hierarchy Level [edit system **scripts commit**]

Release Information Statement introduced in Junos OS Release 7.4.

Description For Junos OS commit scripts, enable transient configuration changes to be committed.

Default Transient changes are disabled by default. If you do not include the **allow-transients** statement, and an enabled script generates transient changes, the command-line interface (CLI) generates an error message and the commit operation fails.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Generating a Persistent or Transient Change on page 159
- Creating a Macro to Read the Custom Syntax and Generate Related Configuration Statements on page 177

apply-macro

Syntax

```
apply-macro apply-macro-name {
    parameter-name parameter-value;
}
```

Hierarchy Level All hierarchy levels

Release Information Statement introduced in Junos OS Release 7.4.

Description With commit script macros, use custom syntax in your configuration.

Macros work by locating **apply-macro** statements that you include in the candidate configuration and using the values specified in the **apply-macro** statement as parameters to a set of instructions (the macro) defined in a commit script. The commit script alters your configuration from one that contains custom syntax into a full configuration containing standard Junos OS statements.

In effect, your custom configuration syntax serves a dual purpose. The syntax allows you to simplify your configuration tasks, and it provides data (or *hooks*) that are used by a commit script macros.

You can include the **apply-macro** statement at any level of the configuration hierarchy. You can include multiple **apply-macro** statements at each level of the configuration hierarchy; however, each must have a unique name.

Options ***apply-macro-name***—Name of the **apply-macro** statement.

parameter-name—One or more parameters. Parameters can be any text you want to include in your configuration.

parameter-value—A value that corresponds to the parameter name. Parameter values can be any text you want to include in your configuration.

Required Privilege Level configure—To enter configuration mode; other required privilege levels depend on where the statement is located in the configuration hierarchy.

Related Documentation

- Overview of Creating Custom Configuration Syntax with Macros on page 171

checksum

Syntax checksum (md5 | sha-256 | sha1) *hash*;

Hierarchy Level [edit event-options **event-script file** *filename*],
[edit system **scripts commit file** *filename*],
[edit system **scripts op file** *filename*]

Release Information Statement introduced in Junos OS Release 9.5.

Description For Junos OS commit scripts and op scripts, specify the MD5, SHA-1, or SHA-256 checksum hash. When it executes a local event, commit, or op script, the Junos OS verifies the authenticity of the script by using the configured checksum hash.

Options

md5 ***hash***—MD5 checksum of this script.

sha-256 ***hash***—SHA-256 checksum of this script.

sha1 ***hash***—SHA-1 checksum of this script.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring Checksum Hashes for a Commit Script on page 192
- Configuring Checksum Hashes for an Event Script on page 450
- Configuring Checksum Hashes for an Op Script on page 319
- Executing an Op Script from a Remote Site on page 321
- file checksum md5 command in the *System Basics and Services Command Reference*
- file checksum sha-256 command in the *System Basics and Services Command Reference*
- file checksum sha1 command in the *System Basics and Services Command Reference*

commit

Syntax

```
commit {
  allow-transients;
  direct-access;
  file filename {
    checksum (md5 | sha-256 | sha1) hash;
    optional;
    refresh;
    refresh-from url;
    source url;
  }
  refresh;
  refresh-from url;
  traceoptions {
    file <filename> <files number> <size size> <world-readable | no-world-readable>;
    flag flag;
    no-remote-trace;
  }
}
```

Hierarchy Level [edit system scripts]

Release Information Statement introduced in Junos OS Release 7.4.

Description For Junos OS commit scripts, configure the commit-time scripting mechanism.

Options The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Implementing Commit Scripts on page 186

direct-access

Syntax direct-access;

Hierarchy Level [edit system scripts commit]

Release Information Statement introduced in Junos OS Release 9.1.

Description Specify that commit scripts read input configurations directly from the database when inspecting these scripts for errors.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Executing Large Commit Scripts on page 193

file (Commit Scripts)

Syntax

```
file filename {
  checksum (md5 | sha-256 | sha1) hash;
  optional;
  refresh;
  refresh-from url;
  source url;
}
```

Hierarchy Level [edit system **scripts commit**]

Release Information Statement introduced in Junos OS Release 7.4.

Description For Junos OS commit scripts, enable a commit script that is located in the **/var/db/scripts/commit** directory.

Options

filename—Name of an Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) file containing a commit script.

The remaining statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Controlling Execution of Commit Scripts During Commit Operations on page 186

optional

Syntax

```
optional;
```

Hierarchy Level [edit system **scripts commit** file *filename*]

Release Information Statement introduced in Junos OS Release 7.4.

Description For Junos OS commit scripts, allow a commit operation to succeed even if the script specified in the **file** statement is missing from the **/var/db/scripts/commit** directory on the device.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Controlling Execution of Commit Scripts During Commit Operations on page 186

refresh (Commit Scripts)

Syntax refresh;

Hierarchy Level [edit system **scripts commit**],
[edit system **scripts commit** file *filename*]

Release Information Statement introduced in Junos OS Release 7.4.

Description For Junos OS commit scripts, overwrite the local copy of all enabled commit scripts or a single enabled script located in the **/var/db/scripts/commit** directory with the copy located at the source URL, as specified in the **source** statement at the same hierarchy level.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Updating a Commit Script from the Master Source on page 191
- **refresh-from (Commit Scripts) on page 298**
- **source (Commit Scripts) on page 300**

refresh-from (Commit Scripts)

Syntax refresh-from *url*;

Hierarchy Level [edit system **scripts commit**],
[edit system **scripts commit** file *filename*]

Release Information Statement introduced in Junos OS Release 7.4.

Description For Junos OS commit scripts, overwrite the local copy of all enabled commit scripts or a single enabled script located in the **/var/db/scripts/commit** directory with the copy located at a URL other than the URL specified in the **source** statement.

Options *url*—The source specified as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Updating a Commit Script from an Alternate Location on page 192
- **refresh (Commit Scripts) on page 298**
- **source (Commit Scripts) on page 300**

scripts

Syntax

```
scripts {
  commit {
    allow-transients;
    direct-access;
    file filename {
      checksum (md5 | sha-256 | sha1) hash;
      optional;
      refresh;
      refresh-from url;
      source url;
    }
    refresh;
    refresh-from url;
    traceoptions {
      file <filename> <files number> <size size> <world-readable | no-world-readable>;
      flag flag;
      no-remote-trace;
    }
  }
  op {
    file filename {
      arguments {
        argument-name {
          description descriptive-text;
        }
      }
      checksum (md5 | sha-256 | sha1) hash;
      command filename-alias;
      description descriptive-text;
      refresh;
      refresh-from url;
      source url;
    }
    refresh;
    refresh-from url;
    traceoptions {
      file <filename> <files number> <size size> <world-readable | no-world-readable>;
      flag flag;
      no-remote-trace;
    }
  }
}
```

Hierarchy Level [edit system]

Release Information Statement introduced in Junos OS Release 7.4.

Description For Junos OS commit or op scripts, configure scripting mechanisms.

Options The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Implementing Commit Scripts on page 186
- Implementing Op Scripts on page 318

source (Commit Scripts)

Syntax source *url*;

Hierarchy Level [edit system **scripts commit file** *filename*]

Release Information Statement introduced in Junos OS Release 7.4.

Description For Junos OS commit scripts, specify the location of the source file for an enabled script located in the **/var/db/scripts/commit** directory. When you include the **refresh** statement at the same hierarchy level and commit the configuration, the local copy is overwritten by the version stored at the specified URL.

Options ***url***—The source specified as an HTTP URL, FTP URL, or scp-style remote file specification.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Configuring the Master Source for a Commit Script on page 191
- Overview of Updating Commit Scripts from a Remote Source on page 189
- **refresh (Commit Scripts) on page 298**
- **refresh-from (Commit Scripts) on page 298**

traceoptions (Commit and Op Scripts)

Syntax

```
traceoptions {
  file <filename> <files number> <size size> <world-readable | no-world-readable>;
  flag flag;
  no-remote-trace;
}
```

Hierarchy Level

[edit system **scripts commit**],
[edit system **scripts op**]

Release Information

Statement introduced in Junos OS Release 7.4.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description

Define tracing operations for commit or op scripts.

Default

If you do not include this statement, no script-specific tracing operations are performed.

Options

filename—Name of the file to receive the output of the tracing operation. All files are placed in the directory **/var/log**. By default, commit script process tracing output is placed in the file **cscript.log** and op script process tracing is placed in the file **op-script.log**. If you include the **file** statement, you must specify a filename. To retain the default, you can specify **cscript.log** or **op-script.log** as the filename.

files ***number***—(Optional) Maximum number of trace files. When a trace file named *trace-file* reaches its maximum size, it is renamed and compressed to *trace-file*.0.gz. When *trace-file* again reaches its maximum size, *trace-file*.0.gz is renamed *trace-file*.1.gz and *trace-file* is renamed and compressed to *trace-file*.0.gz. This renaming scheme continues until the maximum number of trace files is reached. Then the oldest trace file is overwritten.

If you specify a maximum number of files, you also must specify a maximum file size with the **size** option and a filename.

Range: 2 through 1000

Default: 10 files

flag—Tracing operation to perform. To specify more than one tracing operation, include multiple **flag** statements. You can include the following flags:

- **all**—Log all operations
- **events**—Log important events
- **input**—Log script input data
- **offline**—Generate data for offline development
- **output**—Log script output data
- **rpc**—Log script RPCs
- **xslt**—Log the XSLT library

no-world-readable—Restrict file access to owner. This is the default.

size *size*—(Optional) Maximum size of each trace file, in kilobytes (KB), megabytes (MB), or gigabytes (GB). When a trace file named *trace-file* reaches this size, it is renamed and compressed to *trace-file*.0.gz. When *trace-file* again reaches its maximum size, *trace-file*.0.gz is renamed *trace-file*.1.gz and *trace-file* is renamed and compressed to *trace-file*.0.gz. This renaming scheme continues until the maximum number of trace files is reached. Then the oldest trace file is overwritten.

If you specify a maximum file size, you also must specify a maximum number of trace files with the **files** option and a filename.

Syntax: ***x*k** to specify KB, ***x*m** to specify MB, or ***x*g** to specify GB

Range: 10 KB through 1 GB

Default: 128 KB

world-readable—Enable unrestricted file access.

Required Privilege Level

maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Tracing Commit Script Processing on page 195
- Tracing Op Script Processing on page 324

PART 3

Operations Automation

CHAPTER 17

Operation (Op) Scripts Overview

This chapter includes the following topics:

Op Script Programming Overview

Junos OS operation (op) scripts automate network and device management and troubleshooting. Op scripts can perform any function available through the remote procedure calls (RPCs) supported by either the Junos XML management protocol or the Junos Extensible Markup Language (XML) API. Op scripts are executed by the Junos OS management (mgd) process.

Op scripts enable you to do the following things:

- Monitor the overall status of a device.
- Customize the output of operational mode commands.
- Reconfigure the device to avoid or work around known problems in the Junos OS.
- Change the device's configuration in response to a problem.

Op scripts are based on the Junos XML management protocol, and the Junos XML API, which are discussed in "Junos XML API and Junos XML Management Protocol Overview" on page 15. Op scripts can be written in either the Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) scripting language. Op scripts use XPath to locate the operational objects to be inspected and XSLT constructs to specify the actions to perform on the located operational objects. The actions can change the output or execute additional commands based on the output. For more information about XPath and XSLT, see "XPath Overview" on page 21 and "XSLT Overview" on page 19. For more information about SLAX, see "SLAX Overview" on page 35.

How Op Scripts Work

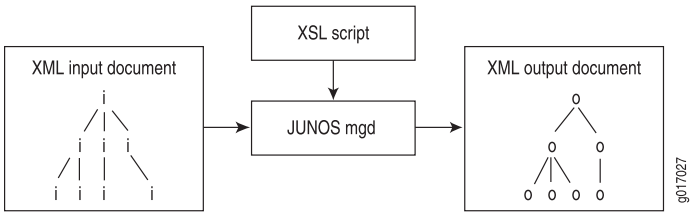
Op scripts execute Junos OS operational commands and inspect the resulting output. After inspection, op scripts can automatically correct errors within the device running Junos OS based on this output.

You add op scripts to device operations by listing the filenames of one or more op script files within the **[edit system scripts op]** hierarchy level. These files must be added to the appropriate op script file directory. For more information about op script file directories, see "Storing Op Scripts in Flash Memory" on page 322. Once added to the device, op scripts are invoked from the command line, using the **op *filename*** command.

You can use op scripts to generate changes to the device configuration by including the **<load-configuration>** tag element. Because the changes are loaded before the standard validation checks are performed, they are validated for correct syntax, just like statements already present in the configuration before the script is applied. If the syntax is correct, the configuration is activated and becomes the active, operational device configuration.

Figure 10 on page 306 shows a high-level view of the flow of op script input and output.

Figure 10: Op Script Input and Output



CHAPTER 18

Writing Op Scripts

This chapter explains how to write operation (op) scripts and includes the following topics:

Required Boilerplate for Op Scripts

When you write operation (op) scripts, you use Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) tools provided with the Junos OS. These tools include basic boilerplate that you must include in all op scripts, optional extension functions that accomplish scripting tasks more easily, and named templates that make scripts easier to read and write, which you import from a file called junos.xsl. For more information about the extension functions and templates, see "Junos Extension Functions in the jcs Namespace" on page 62.

Op scripts are based on Junos XML and Junos XML protocol tag elements. Like all XML elements, angle brackets enclose the name of a Junos XML or Junos XML protocol tag element in its opening and closing tags. This is an XML convention, and the brackets are a required part of the complete tag element name. They are not to be confused with the angle brackets used in the documentation to indicate optional parts of Junos OS CLI command strings.

You must include either XSLT or SLAX boilerplate as the starting point for all op scripts that you create. The XSLT boilerplate follows:

XSLT Boilerplate for Op Scripts

```
1  <?xml version="1.0" standalone="yes"?>
2  <xsl:stylesheet version="1.0"
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4     xmlns:junos="http://xml.juniper.net/junos/*/junos"
5     xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
6     xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
7     <xsl:import href="../import/junos.xsl"/>
```

```
 8     <xsl:template match="/">
 9       <op-script-results>
         <!-- ... insert your code here ... -->
10       </op-script-results>
11     </xsl:template>
     <!-- ... insert additional template definitions here ... -->
12   </xsl:stylesheet>
```

Line 1 is the Extensible Markup Language (XML) processing instruction (PI), which marks this file as XML and specifies the version of XML as 1.0. The XML PI, if present, must be the first non-comment token in the script file.

```
1   <?xml version="1.0"?>
```

Line 2 opens the style sheet and specifies the XSLT version as 1.0.

```
2   <xsl:stylesheet version="1.0"
```

Lines 3 through 6 list all the namespace mappings commonly used in operation scripts. Not all of these prefixes are used in this example, but it is not an error to list namespace mappings that are not referenced. Listing all namespace mappings prevents errors if the mappings are used in later versions of the script.

```
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4     xmlns:junos="http://xml.juniper.net/junos/*/junos"
5     xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
6     xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
```

Line 7 is an XSLT import statement. It loads the templates and variables from the file referenced as **../import/junos.xsl**, which ships as part of the Junos OS (in the file /usr/libdata/cscript/import/junos.xsl). The junos.xsl file contains a set of named templates you can call in your scripts. These named templates are discussed in “Junos Named Templates in the jcs Namespace” on page 82.

```
7     <xsl:import href="../import/junos.xsl"/>
```

Line 8 defines a template that matches the **</>** element. The **<xsl:template match="/">** element is the root element and represents the top level of the XML hierarchy. All XML Path Language (XPath) expressions in the script must start at the top level. This allows the script to access all possible Junos XML and Junos XML protocol remote procedure calls (RPCs). For more information, see “XPath Overview” on page 21.

```
8     <xsl:template match="/">
```

After the **<xsl:template match="/">** tag element, the **<op-script-results>** and **</op-script-results>** container tags must be the top-level child tags, as shown in Lines 9 and 10.

```
9       <op-script-results>
        <!-- ... insert your code here ... -->
10      </op-script-results>
```

Line 11 closes the template.

```
11    </xsl:template>
```

Between Line 11 and Line 12, you can define additional XSLT templates that are called from within the **<xsl:template match="/">** template.

Line 12 closes the style sheet and the op script.

```
12  </xsl:stylesheet>
```

SLAX Boilerplate for Op Scripts

The corresponding SLAX boilerplate is as follows:

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match / {
  <op-script-results> {
    /*
      * Insert your code here
    */
  }
}
```

Mapping Operational Mode Commands and Output Fields to Junos XML Notation

In op scripts, you use tag elements from the Junos XML API to represent operational mode commands and output fields. For the Junos XML equivalent of commands and output fields, consult the *Junos XML API Operational Reference*.

You can also display the Junos XML tag elements for operational mode command output by directing the output from the command to the **| display xml** command:

```
user@host> command-string | display xml
```

For example:

```
user@host> show interfaces terse | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/junos">
    <interface-information
xmlns="http://xml.juniper.net/junos/10.0RI0/junos-interface" junos:style="terse">
        <physical-interface>
            <name>dsc</name>
            <admin-status>up</admin-status>
            <oper-status>up</oper-status>
        </physical-interface>
        <physical-interface>
            <name>fxp0</name>
            <admin-status>up</admin-status>
            <oper-status>up</oper-status>
            <logical-interface>
                <name>fxp0.0</name>
                <admin-status>up</admin-status>
                <oper-status>up</oper-status>
                ...
```

Using RPCs and Operational Mode Commands in Op Scripts

Most Junos OS operational mode commands have XML equivalents. These XML commands can be executed remotely using the *remote procedure call* (RPC) protocol. All operational mode commands that have XML equivalents are listed in the *Junos XML API Operational Reference*.

Use of RPC and operational mode commands in op scripts is discussed in more detail in the following sections:

Using RPCs in Op Scripts

To use an RPC in an op script, include the RPC in a variable declaration. You then invoke the RPC with the **jcs:invoke()** or **jcs:execute()** extension function and include the RPC variable as an argument. The **jcs:invoke()** function executes the RPC on the local device. You can use the **jcs:execute()** function with a connection handle to execute the RPC on a remote device.

The following snippet, which invokes an RPC on the local device, is expanded and fully described in "Example: Customizing Output of the show interfaces terse Command Using an Op Script" on page 340.

XSLT Syntax

```
<xsl:variable name="rpc">
  <get-interface-information/> # Junos RPC for the show interfaces command
</xsl:variable>
<xsl:variable name="out" select="jcs:invoke($rpc)"/>
...
```

SLAX Syntax

```
var $rpc = <get-interface-information>;
var $out = jcs:invoke($rpc);
```

To execute an RPC on a remote device, an SSH session must be established. In order for the script to establish the connection, you must either configure the SSH host key information for the remote device on the local device where the script will be executed, or the SSH host key information for the remote device must exist in the known hosts file of the user executing the script. For each remote device where an RPC is executed, configure the SSH host key information with one of the following methods:

- To configure SSH known hosts on the local device, include the **host** statement, and specify hostname and host key options for the remote device at the **[edit security ssh-known-hosts]** hierarchy level of the configuration.
- To manually retrieve SSH host key information, issue the **set security ssh-known-hosts fetch-from-server *hostname*** configuration mode command to instruct the Junos OS to connect to the remote device and add the key.

```
user@host# set security ssh-known-hosts fetch-from-server router2
```

```
The authenticity of host 'router2 (10.10.10.1)' can't be established.
RSA key fingerprint is 30:18:99:7a:3c:ed:40:04:0f:fd:c1:57:7e:6b:f3:90.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'router2,10.10.10.1' (RSA) to the list of known
 hosts.
```

- To manually import SSH host key information from a file, use the **set security ssh-known-hosts load-key-file *filename*** configuration mode command and specify the known-hosts file.

```
user@host# set security ssh-known-hosts load-key-file /var/tmp/known_hosts

Import SSH host keys from trusted source /var/tmp/known_hosts ? [yes,no]
 (no) yes
```

- Alternatively, the user executing the script can log in to the local device, SSH to the remote device, and then manually accept the host key, which is added to that user's known hosts file. In the following example, root is logged in to **router1**. In order to execute a remote RPC on **router2**, root adds the host key of **router2** by issuing the **ssh router2** operational mode command and manually accepting the key.

```
root@router1> ssh router2

The authenticity of host 'router2 (10.10.10.1)' can't be established.
RSA key fingerprint is 30:18:99:7a:3c:ed:40:04:0f:fd:c1:57:7e:6b:f3:90.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'router2,10.10.10.1' (RSA) to the list of known
 hosts.
```

Displaying the RPC Tags for a Command

To display the remote procedure call (RPC) XML tags for an operational mode command, enter **display xml rpc** after the pipe symbol (|).

The following example displays the RPC tags for the **show route** command:

```
user@host> show route | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.1I0/junos">
    <rpc>
        <get-route-information>
        </get-route-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Using Operational Mode Commands in Op Scripts

Some operational mode commands do not have XML equivalents. If a command is not listed in the *Junos XML API Operational Reference*, the command does not have an XML equivalent.

Another way to determine whether a command has an XML equivalent is to issue the command followed by the **| display xml** command:

```
user@host> operational-mode-command | display xml
```

If the output includes only tag elements like **<output>**, **<cli>**, and **<banner>**, the command might not have an XML equivalent. In the following example, the output indicates that the **show host** command has no XML equivalent:

```
user@host> show host hostname | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/junos">
    <output>
       ...
    </output>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

NOTE: For some commands that have an XML equivalent, the output of the piped **| display xml** command does not include tag elements other than <output>, <cli>, and <banner> only because the relevant feature is not configured. For example, the **show services cos statistics forwarding-class** command has an XML equivalent that returns output in the <service-cos-forwarding-class-statistics> response tag, but if the configuration does not include include any statements at the **[edit class-of-service]** hierarchy level then there is no actual data for the **show services cos statistics forwarding-class | display xml** command to display. The output is something like this:

```
user@host> show services cos statistics forwarding-class | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/8.3I0/junos">

    <cli>

        <banner></banner>

    </cli>

</rpc-reply>
```

For this reason, the information in the *Junos XML API Operational Reference* is normally more reliable.

An op script can include commands that have no XML equivalent. Use the **<command>**, **<xsl:value-of>**, and **<output>** elements in the script, as shown in the following code snippet. This snippet is expanded and fully described in “Example: Displaying DNS Hostname Information Using an Op Script” on page 337.

```
<xsl:variable name="query">
  <command>
    <xsl:value-of select="concat('show host ', $hostname)"/>
  </command>
</xsl:variable>
<xsl:variable name="result" select="jcs:invoke($query)"/>
<xsl:variable name="host" select="$result"/>
<output>
  <xsl:value-of select="concat('Name: ', $host)"/>
</output>
...
```

Declaring Arguments in Op Scripts

There are two ways to declare arguments to an op script: by including XSLT or SLAX instructions in the script or by including statements in the Junos configuration. *Script-generated* and *configuration-generated* arguments have the same operational impact.

To declare arguments within a script, declare a global variable named **arguments**, containing **<argument>** tag elements. Within each **<argument>** tag element, include the required **<name>** tag element and the optional **<description>** tag element:

XSLT Syntax

```
<xsl:variable name="arguments">
  <argument>
    <name>name</name>
    <description>name</description>
  </argument>
</xsl:variable>
```

SLAX Syntax

```
var $arguments = {
  <argument> {
    <name> "name";
    <description> "descriptive-text";
  }
}
```

To declare arguments in the configuration, include the **arguments** statement in the **[edit system scripts op file *filename*]** hierarchy level.

```
[edit system scripts op file filename]
arguments {
  argument-name {
    description descriptive-text;
  }
}
```

If you include the optional **<description>** tag element or the **description** statement, the text of the description appears in the command-line interface (CLI) as a help-text string to describe the purpose of the argument, as discussed in “Configuring Help Text for Op Scripts” on page 315.

In the operation script, you must include a corresponding parameter declaration for each argument. The parameter name must match the name of the argument:

```
<xsl:param name="name"/>
```

The SLAX equivalent is:

```
param $name;
```

You can create a hidden argument by including the **<xsl:param name="*name*"/>** instruction without listing the argument in the **arguments** variable or in the configuration.

After you declare an argument, you can use command completion to list available arguments:

```
user@host> op filename ?
Possible completions:
  argument-name        description
  argument-name        description
```

For each argument you include on the command-line, you must specify a corresponding value. To do this, include an ***argument-name*** and an ***argument-value*** when you execute the script with the **op *filename*** command:

```
user@host> op filename argument-name argument-value
```

NOTE: If you specify an argument that the script does not recognize, the script ignores the argument.

If you configure arguments by including the **arguments** statement in the configuration, any arguments that you declare directly in the script are still available, but are not listed among the **Possible completions** when you issue the **op *filename* ?** command.

If you declare all arguments in the script (and none in the configuration), then the arguments do appear in the **Possible completions** list. This is because the management (mgd) process populates the **Possible completions** list by first checking the configuration for arguments. The mgd process looks in the script for arguments only if no arguments are found in the configuration. Thus, if arguments are declared in the configuration, the arguments declared in the script become hidden in the CLI.

Example: Declaring Arguments

Declare two arguments named **interface** and **protocol**. Execute the script, specifying the **ge-0/2/0.0** interface and the **inet** protocol as values for the arguments. For either method, you must declare corresponding script parameters:

Declaring Arguments in the Op Script (script1)

```
<xsl:param name="interface"/>
<xsl:param name="protocol"/>

<xsl:variable name="arguments">
  <argument>
    <name>interface</name>
    <description>Name of interface to display</description>
  </argument>
  <argument>
    <name>protocol</name>
    <description>Protocol to display (inet, inet6)</description>
  </argument>
</xsl:variable>
```

Declaring Arguments in the Configuration

```
[edit system scripts op]
file script1 {
  arguments {
    interface {
      description "Name of interface to display";
    }
```

```
    protocol {
      description "Protocol to display (inet, inet6)";
    }
  }
}
```

Executing the Script

```
user@host> op script1 interface ge-0/2/0.0 protocol inet
```

Configuring Help Text for Op Scripts

You can provide help text to describe an op script and its arguments when the **?** is used to list possible completions in the CLI. Include the **description** statement:

description *descriptive-text*;

You can include this statement at the following hierarchy levels:

- **[edit system scripts op file** *filename***]**
- **[edit system scripts op file** *filename* **arguments** *argument-name***]**

The following examples show the configuration and the resulting output.

Examples: Configuring Help Text for Op Scripts

Configure help text for a script and display the resulting output:

```
[edit system scripts op]
user@host# set file interface.xsl description "Test the interface"
user@host# commit
...
[edit system scripts op]
user@host# set file ?
Possible completions:
  <name>          Local filename of the script file
  interface.xsl   Test the interface
```

Configure help text for a script's arguments and display the resulting output:

```
[edit system scripts op file interface.xsl arguments]
user@host# set t1 description "Search for T1 interfaces"
user@host# set t3 description "Search for T3 interfaces"
user@host# commit
...
[edit system scripts op file interface.xsl arguments]
user@host# set ?
Possible completions:
  <name>          Name of the argument
  t1              Search for T1 interfaces
  t3              Search for T3 interfaces
```

CHAPTER 19

Configuring and Executing Op Scripts

Operation (op) scripts allow you to automate network troubleshooting and network management. This chapter discusses command-line interface (CLI) configuration statements and operational mode commands for enabling and executing op scripts.

To configure op scripts, include the following statements at the **[edit]** hierarchy level:

```
system {
    scripts {
        op {
            file filename {
                arguments {
                    argument-name {
                        description descriptive-text;
                    }
                }
                checksum (md5 | sha-256 | sha1) hash;
                command filename-alias;
                description descriptive-text;
                refresh;
                refresh-from url;
                source url;
            }
            refresh;
            refresh-from url;
            traceoptions {
                file <filename> <files number> <size size> <world-readable | no-world-readable>;
                flag flag;
                no-remote-trace;
            }
        }
    }
}
```

This chapter discusses the following topics:

Implementing Op Scripts

To use op scripts on a switch, router, or security device, follow these steps:

1. Write the op script. For information about writing op scripts, see "Writing Op Scripts" on page 307. For examples, see "Op Script Examples" on page 329.
2. Copy the script to the /var/db/scripts/op directory on the hard drive or the /config/scripts/op directory on the flash drive. For information about setting the storage location for scripts, see "Storing Op Scripts in Flash Memory" on page 322. Only users who belong to the Junos OS **super-user** login class can access and edit files in these directories.

   NOTE: If the device has dual Routing Engines and you want to enable the op script to execute on both Routing Engines, you must copy the script to the /var/db/scripts/op or /config/scripts/op directory on both Routing Engines. The **commit synchronize** command does not automatically copy scripts between Routing Engines.

3. Enable the script by including the **file *filename*** statement at the **[edit system scripts op]** hierarchy level. For instructions, see "Enabling an Op Script and Defining a Script Alias" on page 318.
4. Issue the **commit** command.

After the commit operation completes, the op script can be executed on the device. See "Executing an Op Script" on page 320.

Enabling an Op Script and Defining a Script Alias

Operation (op) scripts are stored on a device's hard drive in the /var/db/scripts/op directory or on the flash drive in the /config/scripts/op directory. Only users in the Junos OS **super-user** login class can access and edit files in these directories. For information about setting the storage location for scripts, see "Storing Op Scripts in Flash Memory" on page 322.

NOTE: If the device has dual Routing Engines and you want to enable an op script to execute on both Routing Engines, you must copy the script to the /var/db/scripts/op or /config/scripts/op directory on both Routing Engines. The **commit synchronize** command does not automatically copy scripts between Routing Engines.

You must enable an op script before it can be executed. Include the **file *filename*** statement at the **[edit system scripts op]** hierarchy level, specifying the name of an XSLT or SLAX file containing an op script. Only users who belong to the Junos **super-user** login class can enable op scripts.

```
[edit system scripts op]
file filename;
```

The filename of an op script written in SLAX must include the **.slax** extension for the script to be enabled and executed. No particular filename extension is required for op scripts written in XSLT, but we strongly recommend that you append the **.xsl** extension. Whether or not you choose to include the **.xsl** extension on the file, the filename that you add at the **[edit system scripts op]** hierarchy level must exactly match the filename of the script in the directory. For example, if the XSLT script filename is script1.xsl, then you must include script1.xsl in the configuration hierarchy to enable the script; likewise, if the XSLT script filename is script1, then you must include script1 in the configuration hierarchy.

To determine which op scripts are currently active on the device, either list the contents of the /var/run/scripts/op directory or use the **show** command to display the files included at the **[edit system scripts op]** hierarchy level.

Optionally, you can define an alias for an op script and then specify either the filename or the alias when you execute the script. To define the alias, include the **command** statement at the **[edit system scripts op file *filename*]** hierarchy level:

```
[edit system scripts op]
file filename {
  command filename-alias;
}
```

Configuring Checksum Hashes for an Op Script

You can configure one or more checksum hashes that can be used to verify the integrity of an op script before the script runs on the switch, router, or security device.

To configure a checksum hash:

1. Create the script.
2. Place the script in the /var/db/scripts/op directory on the device.
3. Run the script through one or more hash functions to calculate hash values.

   The Junos OS supports MD5, SHA-1, and SHA-256 hash functions.

   ```
   user@host>file checksum md5 /var/db/scripts/commit/script1.slax
     MD5 (/var/db/scripts/op/script1.slax) = 3af7884eb56e2d4489c2e49b26a39a97
   user@host>file checksum sha1 /var/db/scripts/commit/script1.slax
     SHA1 (/var/db/scripts/op/script1.slax) =
     00dc690fb08fb049577d012486c9a6dad34212c0
   user@host>file checksum sha-256 /var/db/scripts/commit/script1.slax
     SHA256 (/var/db/scripts/op/script1.slax) =
     150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
   ```

4. Configure the script with one or more hash values.

   ```
   [edit system scripts op]
   ```

```
user@host#set file script1.slax checksum md5 3af7884eb56e2d4489c2e49b26a39a97
[edit system scripts op]
user@host#set file script1.slax checksum
  sha-1 00dc690fb08fb049577d012486c9a6dad34212c0
[edit system scripts op]
user@host#set file script1.slax checksum
  sha-256 150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
```

During the execution of the script, the Junos OS recalculates the checksum value using the configured hash and verifies that the calculated value matches the configured value. If the values differ, the execution of the script fails. When you configure multiple checksum values with different hash algorithms, all the configured values must match the calculated values; otherwise, the script execution fails.

NOTE: If the op script is stored remotely, do not include the checksum statement in the configuration. You can verify the script's integrity before it runs by specifying the hash value on the command line when you run the op command with the <url> option and the <key> option.

Related Documentation

- Configuring Checksum Hashes for a Commit Script on page 192
- Configuring Checksum Hashes for an Event Script on page 450
- file checksum md5 command in the *System Basics and Services Command Reference*
- file checksum sha-256 command in the *System Basics and Services Command Reference*
- file checksum sha1 command in the *System Basics and Services Command Reference*
- op command in the *System Basics and Services Command Reference*

Executing an Op Script

Unlike commit scripts, operation (op) scripts do not execute during a commit operation. When you issue the **commit** command, op scripts configured at the **[edit system scripts op]** hierarchy level are placed into system memory and enabled for execution. After the commit operation completes, you can execute an op script from the CLI by issuing the **op** operational mode command. You also can configure the device to execute an op script automatically when a member of a specified Junos OS login class logs in to the CLI.

Executing an Op Script by Issuing the op Command

To execute an op script from the CLI, issue the **op** operational mode command, specifying a URL, the script filename, or the alias defined by the **command** statement at the **[edit system scripts op file *filename*]** hierarchy level.

```
user@host> op (filename-or-alias | url url)
```

Executing an Op Script at Login

You can configure an op script to execute automatically when any user belonging to a designated Junos OS login class logs in to the CLI. To associate an op script with a login class, include the **login-script *script-filename*** statement at the **[edit system login class *class-name*]** hierarchy level:

```
[edit system login]
class class-name {
   login-script script-filename;
}
```

The following example configures the super-user-login.slax op script to execute when any user who belongs to the **super-user** class logs in to the CLI (provided that the script has been enabled as discussed in "Enabling an Op Script and Defining a Script Alias" on page 318).

```
[edit system login]
class super-user {
   login-script super-user-login.slax;
}
```

Executing an Op Script from a Remote Site

As an alternative to storing operation (op) scripts locally on the device, you can store op scripts at a remote site. This allows you to execute the scripts by specifying a URL on the command line.

To execute an op script from a remote site:

1. Create the script.
2. (Optional) Store the script temporarily in the /var/tmp directory on the device, and run the script through one or more hash functions to calculate hash values.

   The Junos OS supports MD5, SHA-1, and SHA-256 hash functions.

   ```
   user@host>file checksum md5 /var/tmp/script1.slax
     MD5 (/var/tmp/script1.slax) = 3af7884eb56e2d4489c2e49b26a39a97
   user@host>file checksum sha1 /var/tmp/script1.slax
     SHA1 (/var/tmp/script1.slax) = 00dc690fb08fb049577d012486c9a6dad34212c0
   user@host>file checksum sha-256 /var/tmp/script1.slax
     SHA256 (/var/tmp/script1.slax) =
     150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
   ```
3. Place the script on the remote server.
4. Provide the script URL and the optional hash values to the administrators who will execute the script.
5. Execute the script by running the **op** command and specifying the URL that points to the remote file.

   ```
   user@host>op url https://www.juniper.net/scripts/2009-04-01.01.slax
     key md5 3af7884eb56e2d4489c2e49b26a39a97
   ```

This example shows how to include the <key> option and the MD5 checksum information.

NOTE: If the op script is stored locally, do not include the hash key on the command line. Instead, configure the hash value by including the **checksum** statement at the **[edit system scripts op file *filename*]** hierarchy level. During the execution of the script, the Junos OS recalculates the checksum value using the configured hash and verifies that the calculated value matches the configured value.

Related Documentation

- Configuring Checksum Hashes for an Op Script on page 319
- file checksum md5 command in the *System Basics and Services Command Reference*
- file checksum sha-256 command in the *System Basics and Services Command Reference*
- file checksum sha1 command in the *System Basics and Services Command Reference*
- op command in the *System Basics and Services Command Reference*

Storing Op Scripts in Flash Memory

By default, operation (op) scripts are stored in the /var/db/scripts/op directory on the device's hard drive. To store them in flash memory instead, include the **load-scripts-from-flash** statement at the **[edit system scripts]** hierarchy level:

```
[edit system scripts]
load-scripts-from-flash;
```

The **load-scripts-from-flash** statement applies to all commit, operation, and event scripts; op scripts are stored in the /config/scripts/op directory in flash memory. Changing the scripts' physical location has no effect on their operation.

NOTE: When you add or remove the **load-scripts-from-flash** statement in the configuration, you must manually move scripts from the hard drive to the flash drive, or vice versa, as appropriate. They are not moved automatically.

Specifying a Master Source for an Op Script

You can store a master copy of each op script in a central repository. This eases file management because you can make changes to the master op script in one place and then update the copy on each device where the op script is currently enabled.

To define the location of the master source file for an op script, include the **source** statement at the **[edit system scripts op file *filename*]** hierarchy level:

```
[edit system scripts op file filename]
source url;
```

- ***filename***—Name of the op script.
- ***url***—URL of the op script's master source file. Specify the source as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

The following example specifies an HTTP URL as the remote source for the iso.xsl file:

```
[edit system scripts op]
file iso.xsl {
  source http://my.example.com/pub/scripts/iso.xsl;
}
```

Including the **source** statement in the configuration does not affect the local copy of the op script until you issue the **set refresh** command as described in "Updating an Op Script from the Master Source" on page 323. At that point, the master copy is retrieved from the specified URL and overwrites the local copy.

Updating an Op Script from the Master Source

To update a single op script from its master source, issue the **set refresh** command at the **[edit system scripts op file *filename*]** hierarchy level. The master source must already be configured as described in "Specifying a Master Source for an Op Script" on page 322.

```
[edit system scripts op file filename]
user@host# set refresh
```

To update all enabled op scripts from their master sources, issue the **set refresh** command at the **[edit system scripts op]** hierarchy level:

```
[edit system scripts op]
user@host# set refresh
```

When you issue the **set refresh** command, the switch, router, or security device immediately attempts to connect to the machine that houses the master source for the script files and retrieve a copy of each file. The master copy overwrites the script stored in the local op scripts directory. If a master source is not defined for a script, that script is not updated and a warning is issued.

The update operation occurs as soon as you issue the **set refresh** command. The **refresh** statement is not added to the configuration. In other words, for this statement the **set** command behaves like an operational mode command, instead of adding a statement to the configuration.

If a device has dual Routing Engines and you want the script to be updated on both Routing Engines, you must include the **refresh** statement in the configuration of both Routing Engines. The **commit synchronize** command does not cause the **refresh** statement to take effect on scripts in both Routing Engine directories.

Updating an Op Script from an Alternate Location

In addition to updating an op script from the master source defined by the **source** statement at the **[edit system scripts op file *filename*]** hierarchy level, you also can update a script from an alternate location. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems. To update a single op script from the alternate source, issue the **set refresh-from** command at the **[edit system scripts op file *filename*]** hierarchy level, specifying the location of the remote file:

```
[edit system scripts op file filename]
user@host# set refresh-from url
```

To update all enabled op scripts from the alternate source, issue the **set refresh-from** command at the **[edit system scripts op]** hierarchy level, specifying the location of the remote directory that houses the scripts:

```
[edit system scripts op]
user@host# set refresh-from url
```

At both hierarchy levels:

url—URL of the remote op script or directory. Specify the source as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

When you issue the **set refresh-from** command, the switch, router, or security device attempts to connect to the machine that houses the master source for the script files and retrieve a copy of each file. The master copy overwrites the script stored in the local op scripts directory. If a master source is not defined for a script, that script is not updated and a warning is issued.

The update operation occurs as soon as you issue the **set refresh-from** command. The **refresh-from** statement is not added to the configuration. In other words, for this statement, the **set** command behaves like an operational mode command, instead of adding a statement to the configuration.

If a device has dual Routing Engines and you want the script to be updated on both Routing Engines, you must issue the **set refresh-from** command on each Routing Engine separately. The **commit synchronize** command does not cause the **refresh-from** statement to update scripts on both Routing Engines.

Tracing Op Script Processing

Op script tracing operations track all op script operations and record them in a log file. The logged error descriptions provide detailed information to help you solve problems faster.

The default operation of op script tracing is to log important events in a file called op-script.log located in the /var/log directory. When the file op-script.log reaches 128 kilobytes (KB), it is renamed with a number 0 through 9 (in ascending order) appended to the end of the file and then compressed. The resulting files are op-script.log.0.gz, then op-script.log.1.gz, until there are 10 trace files. Then the oldest trace file (op-script.log.9.gz)

is overwritten. (For more information about how log files are created, see the *Junos OS System Log Messages Reference*.)

This section discusses the following topics:

Minimum Configuration for Enabling Traceoptions for Op Scripts

If no op script trace options are configured, the simplest way to view the trace output of an op script is to configure the **output** trace flag and issue the **show log op-script.log | last** command. To do this, perform the following steps:

1. If you have not done so already, enable an op script by including the **file** statement at the **[edit system scripts op]** hierarchy level:

   ```
   [edit system scripts op]
   user@host# set file filename
   ```

2. Enable trace options by including the **traceoptions flag output** statement at the **[edit system scripts op]** hierarchy level:

   ```
   [edit system scripts op]
   user@host# set traceoptions flag output
   ```

3. Issue the **commit** command:

   ```
   [edit]
   user@host# commit
   ```

4. Display the resulting trace messages recorded in the file */var/log/op-script.log* file. At the end of the log is the output generated by the op script you enabled in Step 1. To display the end of the log, issue the **show log op-script.log | last** operational mode command:

   ```
   [edit]
   user@host# run show log op-script.log | last
   ```

Table 19 on page 325 summarizes useful filtering commands that display selected portions of the **op-script.log** file.

Table 19: Op Script Tracing Operational Mode Commands

Task	Command
Display logging data associated with all op script processing.	**show log op-script.log**
Display processing for only the most recent operation.	**show log op-script.log | last**
Display processing for script errors.	**show log op-script.log | match error**
Display processing for a particular script.	**show log op-script.log | match *filename***

Example: Minimum Configuration for Enabling Traceoptions for Op Scripts

Display the trace output of the op script file **source-route.xsl**:

```
[edit]
system {
   scripts {
      op {
         file source-route.xsl;
         traceoptions flag output;
      }
   }
}
```

```
[edit]
user@host# commit
[edit]
user@host# run show log op-script.log | last
```

Configuring Tracing of Op Scripts

You cannot change the directory (/var/log) to which trace files are written. However, you can customize other trace file settings by including the following statements at the **[edit system scripts op traceoptions]** hierarchy level:

```
[edit system scripts op traceoptions]
file <filename> <files number> <size size> <world-readable | no-world-readable>;
flag all;
flag events;
flag input;
flag offline;
flag output;
flag rpc;
flag xslt;
no-remote-trace;
```

These statements are described in the following sections:

- Configuring the Op Script Log Filename on page 326
- Configuring the Number and Size of Op Script Log Files on page 326
- Configuring Access to Op Script Log Files on page 327
- Configuring the Op Script Trace Operations on page 327

Configuring the Op Script Log Filename

By default, the name of the file that records trace output is op-script.log. You can specify a different name by including the **file** statement at the **[edit system scripts op traceoptions]** hierarchy level:

```
[edit system scripts op traceoptions]
file filename;
```

Configuring the Number and Size of Op Script Log Files

By default, when the trace file reaches 128 KB in size, it is renamed and compressed to *filename*.0.gz, then *filename*.1.gz, and so on, until there are 10 trace files. Then the oldest trace file (*filename*.9.gz) is overwritten.

You can configure the limits on the number and size of trace files by including the following statements at the **[edit system scripts op traceoptions file <*filename*>]** hierarchy level:

```
[edit system scripts op traceoptions file <filename>]
files number size size;
```

For example, set the maximum file size to 640 KB and the maximum number of files to 20. When the file that receives the output of the tracing operation (*filename*) reaches 640 KB, it is renamed and compressed to *filename*.0.gz, and a new file called *filename* is created. When *filename* reaches 640 KB, *filename*.0.gz is renamed *filename*.1.gz and *filename* is renamed and compressed to *filename*.0.gz. This process repeats until there are 20 trace files. Then the oldest file (*filename*.19.gz) is overwritten.

The number of files can range from 2 through 1000 files. The file size can range from 10 KB through 1 gigabyte (GB).

NOTE:

If you set either a maximum file size or a maximum number of trace files, you also must specify the other parameter and a filename.

Configuring Access to Op Script Log Files

By default, access to the op script log file is restricted to the owner. You can manually configure access by including the **world-readable** or **no-world-readable** statement at the **[edit system scripts op traceoptions file <*filename*>]** hierarchy level.

```
[edit system scripts op traceoptions file <filename>]
(world-readable | no-world-readable);
```

The **no-world-readable** statement restricts op script log access to the owner. The **world-readable** statement enables unrestricted access to the op script log file.

Configuring the Op Script Trace Operations

By default, only important events are logged. You can configure the trace operations to be logged by including the following statements at the **[edit system scripts op traceoptions]** hierarchy level:

```
[edit system scripts op traceoptions]
flag all;
flag events;
flag input;
flag offline;
flag output;
flag rpc;
flag xslt;
```

Table 20 on page 327 describes the meaning of the op script tracing flags.

Table 20: Op Script Tracing Flags

Flag	Description	Default Setting
all	Trace all operations.	Off

Table 20: Op Script Tracing Flags *(continued)*

Flag	Description	Default Setting
events	Trace important events.	On
input	Trace op script input data.	Off
offline	Generate data for offline development.	Off
output	Trace op script output data.	Off
rpc	Trace op script RPCs.	Off
xslt	Trace the Extensible Stylesheet Language Transformations (XSLT) library.	Off

CHAPTER 20

Op Script Examples

This chapter provides sample op scripts that run commands and customize output. This chapter includes the following topics:

Changing the Configuration Using Op Scripts

Requirements

This example uses a device running Junos OS.

Overview and Op Script

Op scripts can be used to make structured changes to the Junos OS configuration using the **jcs:load-configuration** template, which is included in the import file junos.xsl. Experienced users, who are familiar with Junos OS, can write scripts that prompt for the relevant configuration information and modify the configuration accordingly. This allows users who have less experience with Junos OS to safely modify the configuration using the script.

When called, the **jcs:load-configuration** template performs the following actions:

1. Locks the configuration database
2. Loads the configuration changes

3. Commits the configuration
4. Unlocks the configuration database

The **jcs:load-configuration** template makes changes to the configuration in **configure exclusive** mode. In this mode, Junos OS locks the candidate *global* configuration for as long as the script accesses the shared database and makes changes to the configuration without interference from other users.

If another user is currently editing the configuration in **configure exclusive** mode or if the database is already locked when the template is called, the call fails. In addition, if there are existing, uncommitted changes to the configuration when the template is called, the commit will fail. If the template call is successful but the commit fails, Junos OS discards the uncommitted changes and rolls back the configuration.

You provide arguments to the **jcs:load-configuration** template to specify how to integrate the changes into the existing configuration, how to customize the commit operation, what changes to make to the configuration, and which connection handle to use. The **call-template** syntax is:

```
<xsl:call-template name="jcs:load-configuration">
  <xsl:with-param name="action" select="(merge | override | replace)"/>
  <xsl:with-param name="commit-options" select="node-set"/>
  <xsl:with-param name="configuration" select="configuration-data"/>
  <xsl:with-param name="connection" select="connection-handle"/>
</xsl:call-template>
```

The following sample SLAX script demonstrates how to use the **jcs:load-configuration** template to disable an interface on a device running Junos OS. All of the values required for the **jcs:load-configuration** template are defined as variables, which are then passed into the template as arguments.

In this example, the **$usage** variable is intialized with a general description of the function of the script. When the script is run, the usage description is output to the CLI using a call to the **jcs:output()** fuction. This allows the user to verify that they are using the script for the correct purpose.

The script calls the **jcs:get-input()** function and prompts the user to enter the name of the interface that should be disabled. The interface name is stored in the **$interface** variable. The configuration data that includes the changes to the configuration are stored in the variable **$config-changes**. This is the value used for the **$configuration** parameter of the **jcs:load-configuration** template. This variable includes the Junos XML API tags for the configuration statements that will be modified. The variable **$interface**, which is supplied by the user, designates the name of the interface to disable.

The **$load-action** variable is initialized to **merge**, which merges the configuration changes in the **$disable** variable with the candidate configuration. This is the equivalent of the CLI configuration mode command **load merge**. Other load options include **replace** and **override**.

The **$options** variable uses the **:=** operator to create a node-set, which is passed to the template as the value of the **$commit-options** parameter. This example includes the **log** tag to add the description of the commit to the commit log file for future reference.

The call to the **jcs:open()** function opens a connection with the Junos OS management process (mgd) and returns a connection handle that is stored in the **$conn_handle** variable. All of the defined variables are passed as arguments to the **jcs:load-configuration** template at the time that it is called.

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns ext = "http://xmlsoft.org/XSLT/namespace";

import "../import/junos.xsl";


match / {
<op-script-results> {

  var $usage = "This script disables the interface specified by the user." _
```

```
                "The script modifies the candidate configuration to disable " _
                "the interface and commits the configuration to activate it.";
    var $temp = jcs:output($usage);

    var $interface = jcs:get-input("Enter interface to disable: ");

    var $config-changes = {
      <configuration> {
        <interfaces> {
          <interface> {
            <name> $interface;
            <disable>;
          }
        }
      }
    }

    var $load-action = "merge";

    var $options := {
      <commit-options> {
        <log> "disabling interface " _ $interface;
      }
    }

    var $conn_handle = jcs:open();

    var $results := {
     call jcs:load-configuration( $action=$load-action, $commit-options=$options,
           $configuration=$config-changes, $connection=$conn_handle);
    }

    $close-results = jcs:close($conn_handle);
  }
  }
```

The **:=** operator copies the results of the **jcs:load-configuration** template call to a temporary variable and runs the **node-set** function on that variable. The resulting node-set is then stored in the **$results** variable. The **:=** operator ensures that the **$results** variable is a node-set rather than a result tree fragment so that the script can access the contents. The **jcs:close()** function closes the connection.

By default, the **jcs:load-configuration** template does not output messages to the CLI. To quickly view any issues with the commit, you should add code to the script to output any error or warning messages that are generated as a result of the **jcs:load-configuration** template call:

```
  if ($results//xnm:error) {
    for-each ($results//xnm:error) {
      <output> message;
    }
  }
  if ($results//xnm:warning) {
    for-each ($results//xnm:warning) {
      <output> message;
```

```
    }
}
```

Device Configuration

Download, enable and run the script:

1. Copy the script contents to a file and save it with a filename that describes its purpose.
2. Download the script to the /var/db/scripts/op/ directory on the device. Only users who belong to the Junos OS **super-user** login class can access and edit files in this directory.
3. In configuration mode, enable the script on the device by adding the script filename to the **[edit system scripts op file]** hierarchy level:

   ```
   user@host# set system scripts op file filename
   ```

4. Note the state of the interface that will be disabled before running the script by issuing the **show interfaces** *interface-name* operational mode command.
5. Run the op script by issuing the **op** *filename* operational mode command:

   ```
   user@host> op filename

   This script disables the interface specified by the user. The script
   modifies the candidate configuration to disable the interface and commits
    the configuration to activate it.
   Enter interface to disable: so-0/0/0

   user@host>
   ```

Verification

Verify the Commit

Purpose Verify that the commit succeeded.

Action You should include code in your script that parses the node-set returned by the **jcs:load-configuration** template for any errors or warnings. This allows you to more easily determine whether the commit succeeded. If there are no warning or error messages, you can verify the success of the commit in several ways.

Check the commit log to verify that the commit was successful. If you included the **log** option in the **$commit-options** parameter, the message should be visible in the commit log along with the commit information.

```
user@host> show system commit

user@host> show system commit
0   2010-09-22 17:08:17 PDT by user via junoscript
    disabling interface so-0/0/0
```

Check the syslog message file to verify that the commit operation was logged. In this case, you also see an **SNMP_TRAP_LINK_DOWN** message for the disabled interface **so-0/0/0**. Depending on your configuration settings for traceoptions, these message may or may not appear in your log file.

```
user@host> file show /var/log/messages | last

Sep 22 17:08:13  host file[7319]: UI_COMMIT: User 'user' requested 'commit'
 operation (comment: disabling interface so-0/0/0)
Sep 22 17:08:16  host xntpd[1386]: ntpd exiting on signal 1
Sep 22 17:08:16  host xntpd[1386]: ntpd 4.2.0-a Fri Jun 25 13:48:13 UTC
2010 (1)
Sep 22 17:08:16  host mib2d[1434]: SNMP_TRAP_LINK_DOWN: ifIndex 526,
ifAdminStatus down(2), ifOperStatus down(2), ifName so-0/0/0
```

Verify the Configuration Changes

Purpose Verify that the correct configuration changes were integrated into the configuration.

Action

- Display the configuration and verify that the changes are visible for the specified interface:

```
user@host> show configuration interfaces so-0/0/0

disable;
```

- For this example, you also can issue the **show interfaces** *interface-name* operational mode command to check that the interface was disabled. In this case, the output captured *before* the interface was disabled shows that the interface is **Enabled**:

```
user@host> show interfaces so-0/0/0

Physical interface: so-0/0/0, Enabled, Physical link is Up
  Interface index: 128, SNMP ifIndex: 526
  Link-level type: PPP, MTU: 4474, Clocking: Internal, SONET mode, Speed:
 OC3, Loopback: None, FCS: 16,
  Payload scrambler: Enabled
  Device flags   : Present Running
  Interface flags: Point-To-Point SNMP-Traps Internal: 0x4000
  Link flags     : Keepalives
  CoS queues     : 4 supported, 4 maximum usable queues
  Last flapped   : 2010-09-14 10:33:25 PDT (1w1d 06:27 ago)
  Input rate     : 0 bps (0 pps)
  Output rate    : 0 bps (0 pps)
  SONET alarms   : None
  SONET defects  : None
```

The output captured *after* running the script to disable the interface shows that the interface is now **Administratively down**:

```
user@host> show interfaces so-0/0/0

brendaw@tp2> show interfaces so-0/0/0
Physical interface: so-0/0/0, Administratively down, Physical link is Up
  Interface index: 128, SNMP ifIndex: 526
  Link-level type: PPP, MTU: 4474, Clocking: Internal, SONET mode, Speed:
 OC3, Loopback: None, FCS: 16,
  Payload scrambler: Enabled
  Device flags   : Present Running
  Interface flags: Down Point-To-Point SNMP-Traps Internal: 0x4000
  Link flags     : Keepalives
  CoS queues     : 4 supported, 4 maximum usable queues
  Last flapped   : 2010-09-14 10:33:25 PDT (1w1d 06:40 ago)
  Input rate     : 0 bps (0 pps)
  Output rate    : 0 bps (0 pps)
```

```
SONET alarms   : None
SONET defects  : None
```

Related Documentation

- Implementing Op Scripts on page 318
- **jcs:close( ) Function on page 63**
- **jcs:get-input( ) Function on page 67**
- **jcs:load-configuration Template on page 86**
- **jcs:open( ) Function on page 70**
- **jcs:output( ) Function on page 71**

Example: Restarting an FPC Using an Op Script

This example simply restarts a Flexible PIC Concentrator (FPC) and slightly modifies the output of the **request chassis fpc** command to include the FPC number that is restarting.

There is no Junos Extensible Markup Language (XML) equivalent for the **request chassis** commands. Therefore, this script uses the **request chassis fpc** command directly rather that using a remote procedure call (RPC). For more information, see "Using RPCs and Operational Mode Commands in Op Scripts" on page 310.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:variable name="arguments">
    <argument>
      <name>slot</name>
      <description>Slot number of the FPC</description>
    </argument>
  </xsl:variable>
  <xsl:param name="slot"/>
  <xsl:template match="/">
    <op-script-results>
      <xsl:variable name="restart">
        <command>
          <xsl:value-of select="concat('request chassis fpc slot ', $slot,'
                                                    restart')"/>
        </command>
      </xsl:variable>
      <xsl:variable name="result" select="jcs:invoke($restart)"/>
      <output>
        <xsl:text>Restarting the FPC in slot </xsl:text>
        <xsl:value-of select="$slot"/>
        <xsl:text>. </xsl:text>
        <xsl:text>To verify, issue the "show chassis fpc" command.</xsl:text>
      </output>
```

```
    </op-script-results>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

var $arguments = {
  <argument> {
    <name> "slot";
    <description> "Slot number of the FPC";
  }
}
param $slot;
match / {
  <op-script-results> {
    var $restart = {
      <command> 'request chassis fpc slot ' _ $slot _ ' restart';
    }
    var $result = jcs:invoke($restart);
    <output> {
      expr "Restarting the FPC in slot ";
      expr $slot;
      expr ". ";
      expr "To verify, issue the \"show chassis fpc\" command.";
    }
  }
}
```

Testing the ex-fpc Script

To test the ex-fpc script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Restarting an FPC Using an Op Script" on page 335 into a text file, name the file ex-fpc.xsl or ex-fpc.slax as appropriate, and copy it to the /var/db/scripts/op directory on the device.
2. In configuration mode, include the **file ex-fpc.*extension*** statement at the **[edit system scripts op]** hierarchy level, substituting **.slax** or **.xsl** for ***extension*** as appropriate.

   ```
   [edit system scripts op]
   file ex-fpc.(slax | xsl);
   ```

3. Issue the **commit and-quit** command.

   ```
   [edit]
   user@host# commit and-quit
   ```

When you issue the **op ex-fpc slot *number*** operational command, you should see output similar to the following:

```
user@host> op ex-fpc slot 0
Restarting the FPC in slot 0. To verify, issue the "show chassis fpc" command.
```

Example: Displaying DNS Hostname Information Using an Op Script

This script displays Domain Name System (DNS) information for a device in your network. The script offers a slight improvement over the **show host *hostname*** command because you do not need to enter a hostname or IP address to view DNS information for the device you are currently using.

There is no Junos Extensible Markup Language (XML) equivalent for the **show host *hostname*** command. Therefore, this script uses the **show host *hostname*** command directly rather that using a remote procedure call (RPC). For more information, see "Using RPCs and Operational Mode Commands in Op Scripts" on page 310.

The script is provided in two distinct versions, one using the **<xsl:choose>** element and the other the **jcs:first-of()** function. Both versions accept the same argument and produce the same output.

XSLT Syntax Using the <xsl:choose> Element

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:variable name="arguments">
    <argument>
      <name>dns</name>
      <description>Name or IP address of a host</description>
    </argument>
  </xsl:variable>
  <xsl:param name="dns"/>
  <xsl:template match="/">
    <op-script-results>
      <xsl:variable name="query">
        <xsl:choose>
          <xsl:when test="$dns">
            <command>
              <xsl:value-of select="concat('show host ', $dns)"/>
            </command>
          </xsl:when>
          <xsl:when test="$hostname">
            <command>
              <xsl:value-of select="concat('show host ', $hostname)"/>
            </command>
          </xsl:when>
        </xsl:choose>
      </xsl:variable>
      <xsl:variable name="result" select="jcs:invoke($query)"/>
      <xsl:variable name="host" select="$result"/>
      <output>
        <xsl:value-of select="concat('Name: ', $host)"/>
      </output>
    </op-script-results>
  </xsl:template>
```

```
</xsl:stylesheet>
```

XSLT Syntax Using the jcs:first-of() Function

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../import/junos.xsl"/>

  <xsl:variable name="arguments">
    <argument>
      <name>dns</name>
      <description>Name or IP address of a host</description>
    </argument>
  </xsl:variable>
  <xsl:param name="dns"/>
  <xsl:template match="/">
    <op-script-results>
      <xsl:variable name="target" select="jcs:first-of($dns, $hostname)"/>
      <xsl:variable name="query">
        <command>
          <xsl:value-of select="concat('show host ', $target)"/>
        </command>
      </xsl:variable>
      <xsl:variable name="result" select="jcs:invoke($query)"/>
      <xsl:variable name="host" select="$result"/>
      <output>
        <xsl:value-of select="concat('Name: ', $host)"/>
      </output>
    </op-script-results>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax Using the <xsl:choose> Element

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

var $arguments = {
  <argument> {
    <name> "dns";
    <description> "Name or IP address of a host";
  }
}
param $dns;
match / {
  <op-script-results> {
    var $query = {
      if ($dns) {
        <command> 'show host ' _ $dns;
      } else if ($hostname) {
        <command> 'show host ' _ $hostname;
      }
```

```
    }
    var $result = jcs:invoke($query);
    var $host = $result;
    <output> 'Name: ' _ $host;
  }
}
```

SLAX Syntax Using the jcs:first-of() Function

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

var $arguments = {
  <argument> {
    <name> "dns";
    <description> "Name or IP address of a host";
  }
}
param $dns;
match / {
  <op-script-results> {
    var $target = jcs:first-of($dns, $hostname);
    var $query = {
      <command> 'show host ' _ $target;
    }
    var $result = jcs:invoke($query);
    var $host = $result;
    <output> 'Name: ' _ $host;
  }
}
```

Testing the ex-hostname Script

To test the ex-hostname script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Displaying DNS Hostname Information Using an Op Script" on page 337 into a text file, name the file ex-hostname.xsl or ex-hostname.slax as appropriate, and copy it to the /var/db/scripts/op directory on the device.
2. In configuration mode, include the **file ex-hostname.*extension*** statement at the **[edit system scripts op]** hierarchy level, substituting **.slax** or **.xsl** for ***extension*** as appropriate.

   ```
   [edit system scripts op]
   file ex-hostname.(slax | xsl);
   ```

3. Issue the **commit and-quit** command.

   ```
   [edit]
   user@host# commit and-quit
   ```

When you issue the **op ex-hostname** operational mode command without the **dns** option, DNS information is displayed for the local device:

```
user@host> op ex-hostname
```

```
Name:
this-router has address 10.168.71.246
```

When you issue the **op ex-hostname dns *hostname*** command, DNS information is displayed for the specified device:

```
user@host> op ex-hostname dns router1
Name:
router1 has address 10.168.71.249
```

When you issue the **op ex-hostname dns *address*** command, DNS information is displayed for the specified address:

```
user@host> op ex-hostname dns 10.168.71.249
Name:
249.71.168.10.IN-ADDR.ARPA domain name pointer router1
```

Example: Customizing Output of the show interfaces terse Command Using an Op Script

By default, the layout of the **show interfaces terse** command looks like this:

```
user@host> show interfaces terse
Interface               Admin Link Proto    Local                 Remote
dsc                     up    up
fxp0                    up    up
fxp0.0                  up    up   inet     192.168.71.246/21
fxp1                    up    up
fxp1.0                  up    up   inet     10.0.0.4/8
                                   inet6    fe80::200:ff:fe00:4/64
                                            fec0::10:0:0:4/64
                                   tnp      4
gre                     up    up
ipip                    up    up
lo0                     up    up
lo0.0                   up    up   inet     127.0.0.1           --> 0/0
lo0.16385               up    up   inet
                                   inet6    fe80::2a0:a5ff:fe12:2f04
lsi                     up    up
mtun                    up    up
pimd                    up    up
pime                    up    up
tap                     up    up
```

In Junos XML, the output fields are represented as follows:

```
user@host> show interfaces terse | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0RI0/junos">
  <interface-information xmlns="http://xml.juniper.net/junos/10.0RI0/junos-interface"
 junos:style="terse">
     <physical-interface>
       <name>dsc</name>
       <admin-status>up</admin-status>
       <oper-status>up</oper-status>
       </physical-interface>
       <physical-interface>
         <name>fxp0</name>
```



```
<admin-status>up</admin-status>
<oper-status>up</oper-status>
<logical-interface>
  <name>fxp0.0</name>
  <admin-status>up</admin-status>
  <oper-status>up</oper-status>
  ... Remainder of output omitted for brevity ...
```

XSLT Syntax The following script customizes the output of the **show interfaces terse** command.

```
1  <?xml version="1.0" standalone="yes"?>
2  <xsl:stylesheet version="1.0"
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4     xmlns:junos="http://xml.juniper.net/junos/*/junos"
5     xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
6     xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
7     <xsl:import href="../import/junos.xsl"/>

8     <xsl:variable name="arguments">
9       <argument>
10        <name>interface</name>
11        <description>Name of interface to display</description>
12      </argument>
13      <argument>
14        <name>protocol</name>
15        <description>Protocol to display (inet, inet6)</description>
16      </argument>
17    </xsl:variable>
18    <xsl:param name="interface"/>
19    <xsl:param name="protocol"/>
20    <xsl:template match="/">
21      <op-script-results>
22        <xsl:variable name="rpc">
23          <get-interface-information>
24            <terse/>
25            <xsl:if test="$interface">
26              <interface-name>
27                <xsl:value-of select="$interface"/>
28              </interface-name>
29            </xsl:if>
30          </get-interface-information>
31        </xsl:variable>
32        <xsl:variable name="out" select="jcs:invoke($rpc)"/>
33        <interface-information junos:style="terse">
34          <xsl:choose>
35            <xsl:when test="$protocol='inet' or $protocol='inet6'
                    or $protocol='mpls' or $protocol='tnp'">
36              <xsl:for-each select="$out/physical-interface/
          logical-interface[address-family/address-family-name = $protocol]">
37                <xsl:call-template name="intf"/>
38              </xsl:for-each>
39            </xsl:when>
40            <xsl:when test="$protocol">
41              <xnm:error>
42                <message>
43                  <xsl:text>invalid protocol: </xsl:text>
```

```
                    <xsl:value-of select="$protocol"/>
                  </message>
                </xnm:error>
              </xsl:when>
              <xsl:otherwise>
                <xsl:for-each select="$out/physical-interface/logical-interface">
                  <xsl:call-template name="intf"/>
                </xsl:for-each>
              </xsl:otherwise>
            </xsl:choose>
          </interface-information>
        </op-script-results>
      </xsl:template>
      <xsl:template name="intf">
        <xsl:variable name="status">
          <xsl:choose>
            <xsl:when test="admin-status='up' and oper-status='up'">
              <xsl:text> </xsl:text>
            </xsl:when>
            <xsl:when test="admin-status='down'">
              <xsl:text>offline</xsl:text>
            </xsl:when>
            <xsl:when test="oper-status='down' and ../admin-status='down'">
              <xsl:text>p-offline</xsl:text>
            </xsl:when>
            <xsl:when test="oper-status='down' and ../oper-status='down'">
              <xsl:text>p-down</xsl:text>
            </xsl:when>
            <xsl:when test="oper-status='down'">
              <xsl:text>down</xsl:text>
            </xsl:when>
            <xsl:otherwise>
              <xsl:value-of select="concat(oper-status, '/', admin-status)"/>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:variable>
        <xsl:variable name="desc">
          <xsl:choose>
            <xsl:when test="description">
              <xsl:value-of select="description"/>
            </xsl:when>
            <xsl:when test="../description">
              <xsl:value-of select="../description"/>
            </xsl:when>
          </xsl:choose>
        </xsl:variable>
        <logical-interface>
          <name><xsl:value-of select="name"/></name>
          <xsl:if test="string-length($desc)">
            <admin-status><xsl:value-of select="$desc"/></admin-status>
          </xsl:if>
          <admin-status><xsl:value-of select="$status"/></admin-status>
          <xsl:choose>
            <xsl:when test="$protocol">
              <xsl:copy-of
              select="address-family[address-family-name = $protocol]"/>
```



```
 99            </xsl:when>
100            <xsl:otherwise>
101              <xsl:copy-of select="address-family"/>
102            </xsl:otherwise>
103          </xsl:choose>
104        </logical-interface>
105      </xsl:template>
106    </xsl:stylesheet>
```

Line-by-Line Explanation of the Script

Lines 1 through 7, Line 20, and Lines 105 and 106 are the boilerplate that you include in every op script. For more information, see “Required Boilerplate for Op Scripts” on page 307.

```
 1  <?xml version="1.0" standalone="yes"?>
 2   <xsl:stylesheet version="1.0"
 3      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 4      xmlns:junos="http://xml.juniper.net/junos/*/junos"
 5      xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
 6      xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
 7      <xsl:import href="../import/junos.xsl"/>
       ...
20      <xsl:template match="/">
         ...
105     </xsl:template>
106    </xsl:stylesheet>
```

Lines 8 through 17 declare a variable called **arguments**, containing two arguments to the script: **interface** and **protocol**. This variable declaration causes **interface** and **protocol** to appear in the command-line interface (CLI) as available arguments to the script.

```
 8      <xsl:variable name="arguments">
 9        <argument>
10          <name>interface</name>
11          <description>Name of interface to display</description>
12        </argument>
13        <argument>
14          <name>protocol</name>
15          <description>Protocol to display (inet, inet6)</description>
16        </argument>
17      </xsl:variable>
```

Lines 18 and 19 declare two parameters to the script, corresponding to the arguments created in Lines 8 through 17. The parameter names must exactly match the argument names.

```
18      <xsl:param name="interface"/>
19      <xsl:param name="protocol"/>
```

Lines 20 through 31 declare a variable named **rpc**. The **show interfaces terse** command is assigned to the **rpc** variable. If you include the **interface** argument when you execute the script, the value of the argument (the interface name) is passed into the script.

```
20      <xsl:template match="/">
21        <op-script-results>
22          <xsl:variable name="rpc">
23            <get-interface-information>
24              <terse/>
```

```
25              <xsl:if test="$interface">
26                <interface-name>
27                  <xsl:value-of select="$interface"/>
28                </interface-name>
29              </xsl:if>
30            </get-interface-information>
31          </xsl:variable>
```

Line 32 declares a variable named **out** and applies to it the execution of the **rpc** variable (**show interfaces terse** command).

```
32          <xsl:variable name="out" select="jcs:invoke($rpc)"/>
```

Line 33 specifies that the output level of the **show interfaces** command being modified is **terse** (as opposed to **extensive**, **detail**, and so on).

```
33          <interface-information junos:style="terse">
```

Lines 34 through 39 specify that if you include the **protocol** argument when you execute the script and if the protocol value that you specify is **inet**, **inet6**, **mpls**, or **tnp**, the **intf** template is applied to each instance of that protocol type in the output.

```
34            <xsl:choose>
35              <xsl:when test="$protocol='inet' or $protocol='inet6'
                or $protocol='mpls' or $protocol='tnp'">
36                <xsl:for-each select="$out/physical-interface/
          logical-interface[address-family/address-family-name = $protocol]">
37                  <xsl:call-template name="intf"/>
38                </xsl:for-each>
39              </xsl:when>
```

Lines 40 through 47 specify that if you include the **protocol** argument when you execute the script and if the protocol value that you specify is something other than **inet**, **inet6**, **mpls**, or **tnp**, an error message is emitted.

```
40              <xsl:when test="$protocol">
41                <xnm:error>
42                  <message>
43                    <xsl:text>invalid protocol: </xsl:text>
44                    <xsl:value-of select="$protocol"/>
45                  </message>
46                </xnm:error>
47              </xsl:when>
```

Lines 48 through 52 specify that if you do not include the **protocol** argument when you execute the script, the **intf** template is applied to each logical interface in the output.

```
48              <xsl:otherwise>
49                <xsl:for-each select="$out/physical-interface/logical-interface">
50                  <xsl:call-template name="intf"/>
51                </xsl:for-each>
52              </xsl:otherwise>
```

Lines 53 through 56 are closing tags.

```
53            </xsl:choose>
54          </interface-information>
55        </op-script-results>
56      </xsl:template>
```

Line 57 opens the **intf** template. This template customizes the output of the **show interfaces terse** command.

```
57    <xsl:template name="intf">
```

Line 58 declares a variable called **status**, the purpose of which is to specify how the interface status is reported. Lines 59 through 79 contain a **<xsl:choose>** instruction that populates the **status** variable by considering all the possible states. As always in XSLT, the first **<xsl:when>** instruction that evaluates as TRUE is executed, and the remainder are ignored. Each **<xsl:when>** instruction is explained separately.

```
58      <xsl:variable name="status">
59        <xsl:choose>
```

Lines 60 through 62 specify that if **admin-status** is **up** and **oper-status** is **up**, no output is generated. In this case, the **status** variable remains empty.

```
60          <xsl:when test="admin-status='up' and oper-status='up'">
61            <xsl:text> </xsl:text>
62          </xsl:when>
```

Lines 63 through 65 specify that if **admin-status** is **down**, the **status** variable contains the text **offline**.

```
63          <xsl:when test="admin-status='down'">
64            <xsl:text>offline</xsl:text>
65          </xsl:when>
```

Lines 66 through 68 specify that if **oper-status** is **down** and the physical interface **admin-status** is **down**, the **status** variable contains the text **p-offline**. (**../** selects the physical interface.)

```
66          <xsl:when test="oper-status='down' and ../admin-status='down'">
67            <xsl:text>p-offline</xsl:text>
68          </xsl:when>
```

Lines 69 through 71 specify that if **oper-status** is **down** and the physical interface **oper-status** is **down**, the **status** variable contains the text **p-down**. (**../** selects the physical interface.)

```
69          <xsl:when test="oper-status='down' and ../oper-status='down'">
70            <xsl:text>p-down</xsl:text>
71         </xsl:when>
```

Lines 72 through 74 specify that if **oper-status** is **down**, the **status** variable contains the text **down**.

```
72          <xsl:when test="oper-status='down'">
73            <xsl:text>down</xsl:text>
74          </xsl:when>
```

Lines 75 through 77 specify that if none of the test cases are true, the **status** variable contains **oper-status** and **admin-status** concatenated with a slash as a separator.

```
75          <xsl:otherwise>
76            <xsl:value-of select="concat(oper-status, '/', admin-status)"/>
77          </xsl:otherwise>
```

Lines 78 and 79 are closing tags.

```
78        </xsl:choose>
```

```
79      </xsl:variable>
```

Lines 80 through 89 define a variable called **desc**. An **<xsl:choose>** instruction populates the variable by selecting the most specific interface description available. If a logical interface description is included in the configuration, it is used to populate the **desc** variable. If not, the physical interface description is used. If no physical interface description is included in the configuration, the variable remains empty. As always in XSLT, the first **<xsl:when>** instruction that evaluates as TRUE is executed, and the remainder are ignored.

```
80      <xsl:variable name="desc">
81        <xsl:choose>
82          <xsl:when test="description">
83            <xsl:value-of select="description"/>
84          </xsl:when>
85          <xsl:when test="../description">
86            <xsl:value-of select="../description"/>
87          </xsl:when>
88        </xsl:choose>
89      </xsl:variable>
```

The remainder of the script specifies how the operational mode output is displayed.

Lines 90 and 91 specify that the logical interface name is displayed first in the output.

```
90      <logical-interface>
91        <name><xsl:value-of select="name"/></name>
```

Lines 92 through 94 test whether the **desc** variable has a nonzero number of characters. If the number of characters is more than zero, the interface description is displayed in the standard location of the **admin-status** field. (In standard output, the **admin-status** field is displayed on the second line.)

```
92        <xsl:if test="string-length($desc)">
93          <admin-status><xsl:value-of select="$desc"/></admin-status>
94        </xsl:if>
```

Line 95 specifies that the interface status as defined in the **status** variable is displayed next.

```
95        <admin-status><xsl:value-of select="$status"/></admin-status>
```

Lines 96 through 103 specify that if you include the **protocol** argument when you execute the script, only interfaces with that protocol configured are displayed. If you do not include the **protocol** argument, all interfaces are displayed.

```
96        <xsl:choose>
97          <xsl:when test="$protocol">
98            <xsl:copy-of
              select="address-family[address-family-name = $protocol]"/>
99          </xsl:when>
100         <xsl:otherwise>
101           <xsl:copy-of select="address-family"/>
102         </xsl:otherwise>
103       </xsl:choose>
```

Lines 104 through 106 are closing tags.

```
104     </logical-interface>
105   </xsl:template>
```

```
106   </xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

var $arguments = {
  <argument> {
    <name> "interface";
    <description> "Name of interface to display";
  }
  <argument> {
    <name> "protocol";
    <description> "Protocol to display (inet, inet6)";
  }
}
param $interface;
param $protocol;
match / {
  <op-script-results> {
    var $rpc = {
      <get-interface-information> {
        <terse>;
        if ($interface) {
          <interface-name> $interface;
        }
      }
    }
    var $out = jcs:invoke($rpc);
    <interface-information junos:style="terse"> {
      if ($protocol='inet' or $protocol='inet6' or $protocol='mpls' or
                $protocol='tnp') {
        for-each ($out/physical-interface/
            logical-interface[address-family/address-family-name = $protocol]) {
          call intf();
        }
      } else if ($protocol) {
        <xnm:error> {
          <message> {
            expr "invalid protocol: ";
            expr $protocol;
          }
        }
      } else {
        for-each ($out/physical-interface/logical-interface) {
          call intf();
        }
      }
    }
  }
}
intf () {
  var $status = {
    if (admin-status='up' and oper-status='up') {
```

```
    } else if (admin-status='down') {
      expr "offline";
    } else if (oper-status='down' and ../admin-status='down') {
      expr "p-offline";
    } else if (oper-status='down' and ../oper-status='down') {
      expr "p-down";
    } else if (oper-status='down') {
      expr "down";
    } else {
      expr oper-status _ '/' _ admin-status;
    }
  }
  var $desc = {
    if (description) {
      expr description;
    } else if (../description) {
      expr ../description;
    }
  }
  <logical-interface> {
    <name> name;
    if (string-length($desc)) {
      <admin-status> $desc;
    }
    <admin-status> $status;
    if ($protocol) {
      copy-of address-family[address-family-name = $protocol];
    } else {
      copy-of address-family;
    }
  }
}
```

Testing the ex-interface Script

To test the ex-interface script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Customizing Output of the show interfaces terse Command Using an Op Script" on page 340 into a text file, name the file ex-interface.xsl or ex-interface.slax as appropriate, and copy it to the /var/db/scripts/op directory on the device.
2. In configuration mode, include the **file ex-interface.*extension*** statement at the **[edit system scripts op]** hierarchy level, substituting **.slax** or **.xsl** for ***extension*** as appropriate.

   ```
   [edit system scripts op]
   file ex-interface.(slax | xsl);
   ```

3. Issue the **commit and-quit** command.

   ```
   [edit]
   user@host# commit and-quit
   ```

Issue the **show interfaces terse** and **op ex-interface** operational commands and compare the output.

```
user@host> show interfaces terse
Interface               Admin Link Proto    Local                 Remote
dsc                     up    up
fxp0                    up    up
fxp0.0                  up    up   inet     192.168.71.246/21
fxp1                    up    up
fxp1.0                  up    up   inet     10.0.0.4/8
                                   inet6    fe80::200:ff:fe00:4/64
                                            fec0::10:0:0:4/64
                                   tnp      4
gre                     up    up
ipip                    up    up
lo0                     up    up
lo0.0                   up    up   inet     127.0.0.1           --> 0/0
lo0.16385               up    up   inet
                                   inet6    fe80::2a0:a5ff:fe12:2f04
lsi                     up    up
mtun                    up    up
pimd                    up    up
pime                    up    up
tap                     up    up


user@host> op ex-interface
Interface               Admin Link Proto    Local                 Remote
fxp0.0                  This is the Ethernet Management interface.
                                   inet     192.168.71.246/21
fxp1.0                             inet     10.0.0.4/8
                                   inet6    fe80::200:ff:fe00:4/64
                                            fec0::10:0:0:4/64
                                   tnp      4
lo0.0                              inet     127.0.0.1           --> 0/0
lo0.16385                          inet
                                   inet6    fe80::2a0:a5ff:fe12:2f04-->


user@host> op ex-interface interface fxp0
Interface               Admin Link Proto    Local                 Remote
fxp0.0                  This is the Ethernet Management interface.
                                   inet     192.168.71.246/21


user@host> op ex-interface protocol inet
Interface               Admin Link Proto    Local                 Remote
fxp0.0                  This is the Ethernet Management interface.
                                   inet     192.168.71.246/21
fxp1.0                             inet     10.0.0.4/8
lo0.0                              inet     127.0.0.1           --> 0/0
lo0.16385                          inet
```

Example: Finding LSPs to Multiple Destinations Using an Op Script

This sample script checks for label-switched paths (LSPs) to multiple destinations.

XSLT Syntax

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:junos="http://xml.juniper.net/junos/*/junos"
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0" version="1.0">
```

```
<xsl:variable name="arguments">
  <argument>
    <name>address</name>
    <description>LSP endpoint</description>
  </argument>
</xsl:variable>
<xsl:param name="address"/>
<xsl:template match="/">
  <op-script-output>
    <xsl:choose>
      <xsl:when test="$address = ''">
        <xnm:error>
          <message>missing mandatory argument 'address'</message>
        </xnm:error>
      </xsl:when>
      <xsl:otherwise>
        <xsl:variable name="get-configuration">
          <get-configuration database="committed">
            <configuration>
              <protocols>
                <mpls/>
              </protocols>
            </configuration>
          </get-configuration>
        </xsl:variable>
        <xsl:variable name="config"
              select="jcs:invoke($get-configuration)"/>
        <xsl:variable name="mpls" select="$config/protocols/mpls"/>
        <xsl:variable name="get-route-information">
          <get-route-information>
            <terse/>
            <destination>
              <xsl:value-of select="$address"/>
            </destination>
          </get-route-information>
        </xsl:variable>
        <xsl:variable name="rpc-out"
                select="jcs:invoke($get-route-information)"/>
        <xsl:choose>
          <xsl:when test="$rpc-out//xnm:error">
            <xsl:copy-of select="$rpc-out//xnm:error"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:for-each select="$rpc-out/route-table/rt/rt-destination">
              <xsl:choose>
                <xsl:when test="contains(.,'/32')">
                  <xsl:variable name="dest"
                        select="substring-before(.,'/')"/>
                  <xsl:variable name="lsp"
                      select="$mpls/label-switched-path[to = $dest]"/>
                  <xsl:choose>
                    <xsl:when test="$lsp">
                      <output>
                        <xsl:value-of select="concat('Found: ', $dest,
                                ' (',$lsp/to, ') --&gt; ', $lsp/name)"/>
```



```
                                    </output>
                                  </xsl:when>
                                  <xsl:otherwise>
                                    <xsl:variable name="name"
                                                  select="jcs:hostname($dest)"/>
                                    <output>
                                      <xsl:value-of select="concat('Name: ', $name)"/>
                                    </output>
                                    <output>
                                      <xsl:value-of select="concat('Missing: ', $dest)"/>
                                    </output>
                                  </xsl:otherwise>
                                </xsl:choose>
                              </xsl:when>
                              <xsl:otherwise>
                                <output>
                                  <xsl:value-of select="concat('Not a host route: ', .)"/>
                                </output>
                              </xsl:otherwise>
                            </xsl:choose>
                          </xsl:for-each>
                        </xsl:otherwise>
                      </xsl:choose>
                    </xsl:otherwise>
                  </xsl:choose>
                </op-script-output>
              </xsl:template>
            </xsl:stylesheet>
```

SLAX Syntax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

var $arguments = {
  <argument> {
    <name> "address";
    <description> "LSP endpoint";
  }
}
param $address;
match / {
  <op-script-output> {
    if ($address = '') {
      <xnm:error> {
        <message> "missing mandatory argument 'address'";
      }
    } else {
      var $get-configuration = {
        <get-configuration database="committed"> {
          <configuration> {
            <protocols> {
              <mpls>;
            }
          }
        }
```

```
    }
    var $config = jcs:invoke($get-configuration);
    var $mpls = $config/protocols/mpls;
    var $get-route-information = {
      <get-route-information> {
        <terse>;
        <destination> $address;
      }
    }
    var $rpc-out = jcs:invoke($get-route-information);
    if ($rpc-out//xnm:error) {
      copy-of $rpc-out//xnm:error;
    } else {
      for-each ($rpc-out/route-table/rt/rt-destination) {
        if (contains(.,'/32')) {
          var $dest = substring-before(.,'/');
          var $lsp = $mpls/label-switched-path[to = $dest];
          if ($lsp) {
            <output> 'Found: ' _ $dest _ ' (' _ $lsp/to _ ') --> ' _
                $lsp/name;
          } else {
            var $name = jcs:hostname($dest);
            <output> 'Name: ' _ $name;
            <output> 'Missing: ' _ $dest;
          }
        } else {
          <output> 'Not a host route: ' _ .;
        }
      }
    }
  }
}
```

Testing the ex-lsp Script

To test the ex-lsp script, perform the following steps:

1. Copy the XSLT or SLAX script from "Example: Finding LSPs to Multiple Destinations Using an Op Script" on page 349 into a text file, name the file ex-lsp.xsl or ex-lsp.slax as appropriate, and copy it to the /var/db/scripts/op directory on the device.
2. In configuration mode, include the **file ex-lsp.*extension*** statement at the **[edit system scripts op]** hierarchy level, substituting **.slax** or **.xsl** for ***extension*** as appropriate.

   ```
   [edit system scripts op]
   file ex-lsp.(slax | xsl);
   ```

3. Issue the **commit and-quit** command.

   ```
   [edit]
   user@host# commit and-quit
   ```

When you issue the **op ex-lsp address *address*** operational mode command, you should see output similar to the following:

```
user@R4> op ex-lsp address 10.168.215.0/24
Found: 192.168.215.1 (10.168.215.1) --> R4>R1
Found: 192.168.215.2 (10.168.215.2) --> R4>R2
Name: R3
Missing: 10.168.215.3
Name: R5
Missing: 10.168.215.4
Name: R6
Missing: 10.168.215.5
```

Example: Importing and Exporting Files Using an Op Script

Use the Junos XML protocol **file-put** and **file-get** operations to move a file to or from a remote server. Especially useful when the remote server sits outside a firewall, these commands move the files using the existing Junos XML protocol stream. This obviates the need to open a separate stream for the file transfer, handle authentication before moving files, and verify that the file transfer service is available.

Exporting Files to a Remote Server

Use the Junos XML protocol **file-put** operation to transfer files within an existing remote Junos XML protocol connection. The basic syntax for using the **file-put** command is as follows:

```
<rpc>
  <file-put>
    <filename>value</filename>
    <encoding>value</encoding>
    <permission>value</permission>
    <delete-if-exist />
    <file-contents>file</file-contents>
  </file-put>
</rpc>
```

The following attributes are used with the **file-put** command. These attributes can be placed in any order with the exception of the **file-contents** attribute. The **file-contents** attribute must be the last attribute used with the **file-put** command.

- **filename**—(Mandatory) Within this tag, you include the full or relative file path and filename of the file for export. When you use a relative file path, the specified file path should be relative to the user's home directory. If the specified file directory does not exist, the system returns a "directory not found" error.
- **encoding**—(Mandatory) Specifies the type of encoding used. You can use **ASCII** or **base64** encoding.
- **permission**—(Optional) Sets the file's UNIX permission on the remote server. For example, to apply read/write access for the user, and read access to others, you would set the permission value to 0644. For a full explanation of UNIX permissions, see the **chmod** command.

- **delete-if-exist**—(Optional) If specified, an existing file on the remote server will be overwritten. If this attribute is not set, an error is returned if an existing file is encountered.
- **file-contents**—(Mandatory) The **ASCII** or **base64** encoded file to be exported to the remote server. This attribute must be the last attribute used.

Importing Files from a Remote Server

The Junos XML protocol **file-get** operation can be used to transfer files within an existing remote Junos XML protocol connection. Unless otherwise specified by the user, the imported file will be placed in the user's home directory. The basic syntax for using the **file-get** command is as follows:

```
<rpc>
  <file-get>
    <filename>value</filename>
    <encoding>value</encoding>
  </file-put>
</rpc>
```

The following attributes are used with the **file-get** command.

- **filename**—(Mandatory) Within this tag, you include the full or relative file path and filename of the file for import. When you use a relative file path, the specified file path is relative to the user's home directory.
- **encoding**—(Mandatory) Specifies the type of encoding used. You can use ASCII or base64 encoding.

NOTE: When you use ASCII encoding, the file-get operation converts any control characters in the imported file to the Unicode character 'SECTION SIGN' (U+00A7).

CHAPTER 21

Summary of Op Script Configuration Statements

This chapter describes each configuration statement for operation (op) scripts. The statements are organized alphabetically.

arguments

Syntax

```
arguments {
  argument-name {
    description descriptive-text;
  }
}
```

Hierarchy Level [edit system **scripts op file** *filename*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, configure command-line arguments to the script.

Options ***argument-name***—The name of a command-line argument to an op script.

The remaining statement is explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Declaring Arguments in Op Scripts on page 313

checksum

Syntax checksum (md5 | sha-256 | sha1) *hash*;

Hierarchy Level [edit event-options **event-script file** *filename*],
[edit system **scripts commit file** *filename*],
[edit system **scripts op file** *filename*]

Release Information Statement introduced in Junos OS Release 9.5.

Description For Junos OS commit scripts and op scripts, specify the MD5, SHA-1, or SHA-256 checksum hash. When it executes a local event, commit, or op script, the Junos OS verifies the authenticity of the script by using the configured checksum hash.

Options

md5 *hash*—MD5 checksum of this script.

sha-256 *hash*—SHA-256 checksum of this script.

sha1 *hash*—SHA-1 checksum of this script.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring Checksum Hashes for a Commit Script on page 192
- Configuring Checksum Hashes for an Event Script on page 450
- Configuring Checksum Hashes for an Op Script on page 319
- Executing an Op Script from a Remote Site on page 321
- file checksum md5 command in the *System Basics and Services Command Reference*
- file checksum sha-256 command in the *System Basics and Services Command Reference*
- file checksum sha1 command in the *System Basics and Services Command Reference*

command

Syntax command *filename-alias*;

Hierarchy Level [edit system **scripts op file** *filename*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, configure a filename alias for the script file. This allows you to run the script by referencing either the script filename or the filename alias.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Enabling an Op Script and Defining a Script Alias on page 318

description

Syntax description *descriptive-text*;

Hierarchy Level [edit system **scripts op file** *filename*]
[edit system **scripts op file** *filename* **arguments** *argument-name*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, provide a help-text string that appears in the command-line interface (CLI).

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Configuring Help Text for Op Scripts on page 315
- Declaring Arguments in Op Scripts on page 313
- **file (Op Scripts) on page 358**

file (Op Scripts)

Syntax

```
file filename {
  arguments {
    argument-name {
      description descriptive-text;
    }
  }
  checksum (md5 | sha-256 | sha1) hash;
  command filename-alias;
  description descriptive-text;
  refresh;
  refresh-from url;
  source url;
}
```

Hierarchy Level [edit system **scripts op**]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, enable an op script that is located in the **/var/db/scripts/op** directory.

Options ***filename***—The name of an Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) file containing an op script.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Enabling an Op Script and Defining a Script Alias on page 318

op

Syntax

```
op {
  file filename {
    arguments {
      argument-name {
        description descriptive-text;
      }
    }
    checksum (md5 | sha-256 | sha1) hash;
    command filename-alias;
    description descriptive-text;
    refresh;
    refresh-from url;
    source url;
  }
  refresh;
  refresh-from url;
  traceoptions {
    file <filename> <files number> <size size> <world-readable | no-world-readable>;
    flag flag;
    no-remote-trace;
  }
}
```

Hierarchy Level [edit system scripts]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, configure an operation scripting mechanism.

Options The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Implementing Op Scripts on page 318

refresh (Op Scripts)

Syntax refresh;

Hierarchy Level [edit system **scripts op**],
[edit system **scripts op file** *filename*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, overwrite the local copy of all enabled op scripts or a single enabled script located in the **/var/db/scripts/op** directory with the copy located at the source URL, specified in the **source** statement at the same hierarchy level.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Updating an Op Script from the Master Source on page 323
- **refresh-from (Op Scripts) on page 360**
- **source (Op Scripts) on page 361**

refresh-from (Op Scripts)

Syntax refresh-from *url*;

Hierarchy Level [edit system **scripts op**],
[edit system **scripts op file** *filename*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, overwrite the local copy of all enabled op scripts or a single enabled script located in the **/var/db/scripts/op** directory with the copy located at a URL other than the URL specified in the **source** statement.

Options *url*—Source specified as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Updating an Op Script from an Alternate Location on page 324
- **refresh (Op Scripts) on page 360**
- **source (Op Scripts) on page 361**

scripts

See **scripts**

source (Op Scripts)

Syntax source *url*;

Hierarchy Level [edit system **scripts op file** *filename*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, specify the location of the source file for an enabled script located in the **/var/db/scripts/op** directory. When you include the **refresh** statement at the same hierarchy level, the local copy is overwritten by the version stored at the specified URL.

Options ***url***—Master source file for an op script specified as an HTTP URL, FTP URL, or scp-style remote file specification.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Specifying a Master Source for an Op Script on page 322
- Updating an Op Script from the Master Source on page 323
- **refresh (Op Scripts) on page 360**
- **refresh-from (Op Scripts) on page 360**

traceoptions

See **traceoptions (Commit and Op Scripts)**

PART 4

Event Policy

CHAPTER 22

Event Policy Overview

This chapter discusses the following topics:

Event Notifications and Policies Overview

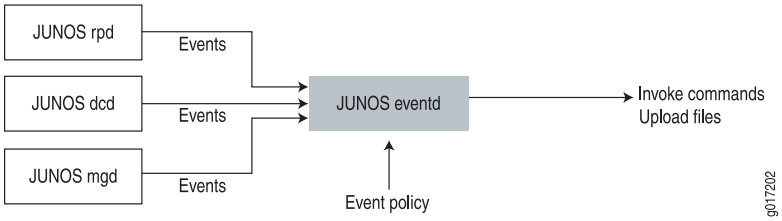
To diagnose a fault or error condition on a device, you need relevant information about the state of the platform. You can derive state information from *event notifications*. Event notifications are system log messages and SNMP traps. A Junos OS process called the *event process* (eventd) receives event notifications—henceforth simply called *events*—from other Junos OS processes.

Timely diagnosis and intervention can correct error conditions and keep the device in operation. After the eventd process receives events, *event policies* instruct the eventd process to select specific events, correlate the events, and perform a set of actions. These actions can either help you diagnose a fault or take corrective action. For example, the eventd process can upload device files to a given destination and issue operational mode commands.

Events can originate as SNMP traps or system log messages. The event process receives event messages from other Junos OS processes, such as the routing protocol process (rpd) and the management process (mgd). Depending on the custom event policy you configure, eventd listens for specific events and in response to these events might create a log file, invoke a Junos command, or invoke an event script. When an event script is invoked, event details are passed to the event script in the form of XML inputs.

Figure 11 on page 365 shows how the event process (eventd) interacts with other Junos OS processes.

Figure 11: Interaction of eventd Process with Other Junos OS Processes



How Event Policies Work

An event policy is an if-then-else construct. It defines actions to be executed by the eventd process on receipt of an event. You can configure multiple policies to be processed for an event. The policies are executed in the order in which they appear in the configuration. For each policy, you can configure multiple actions. The actions are also executed in the order in which they appear in the configuration.

To view a list of the events that can be referenced in an event policy, issue the **help syslog ?** command:

```
user@host> help syslog ?
Possible completions:
  <syslog-tag>         System log tag
  ACCT_ACCOUNTING_FERROR  Error occurred during file processing
  ACCT_ACCOUNTING_FOPEN_ERROR  Open operation failed on file
  ...
```

You can filter the output of a search by using the pipe (**|**) symbol. The following example lists the filters that can be used with the pipe symbol:

```
user@host> help syslog | ?
Possible completions:
  count                Count occurrences
  display              Show additional kinds of information
  except               Show only text that does not match a pattern
  find                 Search for first occurrence of pattern
  hold                 Hold text without exiting the --More-- prompt
  last                 Display end of output only
  match                Show only text that matches a pattern
  no-more              Don't paginate output
  request              Make system-level requests
  resolve              Resolve IP addresses
  save                 Save output text to file
  trim                 Trim specified number of columns from start of line
```

For more information about using the pipe symbol, see the *Junos OS CLI User Guide*.

In response to events, the eventd process can correlate two or more events based on a policy, and execute the following actions:

- Ignore the event—Do not generate a system log message for this event and do not process any further policy instructions for this event.
- Upload a file—Upload a file to a specified destination. You can specify a transfer delay, so that, on receipt of an event, the upload of the file begins after the configured transfer delay. For example, to upload a core file, a transfer delay can ensure that the core file has been completely generated before the upload begins.
- Execute Junos OS operational mode commands—Execute commands on receipt of an event. The XML or text output of these commands is stored in a file, which is then uploaded to a specified URL. You can include variables in the command that allow data from the triggering event to be automatically included in the command syntax.

- Execute Junos OS event scripts—Execute event scripts on receipt of an event. Event scripts are Extensible Stylesheet Transformation (XSLT) or Stylesheet Language Alternative Syntax (SLAX) scripts that you write to perform any function available through Junos XML or Junos XML protocol remote procedure calls (RPCs). Additionally, you can pass to an event script a set of arguments that you define. A script can build and run an operational mode command, receive the command output, inspect the output, and determine the next appropriate action. This process can be repeated until the source of the problem is determined. The output of the scripts is stored in a file, which is then uploaded to a specified URL. You can include variables in the arguments to the scripts that allow data from the triggering event to be incorporated into the script.
- Raise an SNMP trap.

CHAPTER 23

Configuring Event Policy

Event policies can monitor specific events, create log files, invoke Junos OS commands, and invoke event scripts. This chapter discusses the command-line interface (CLI) statements for configuring event policies.

To configure event policy, include the following statements at the **[edit event-options]** hierarchy level:

```
[edit event-options]
destinations {
    destination-name {
        archive-sites {
            url <password password>;
        }
        transfer-delay seconds;
    }
}
generate-event event-name {
    time-interval seconds;
    time-of-day hh:mm:ss;
}
policy policy-name {
    attributes-match {
        event1.attribute-name equals event2.attribute-name;
        event.attribute-name matches regular-expression;
        event1.attribute-name starts-with event2.attribute-name;
    }
    events [ events ];
    within seconds {
        events [ events ];
        not events [ events ];
        trigger (on | after | until) event-count;
    }
    then {
        event-script filename {
            arguments {
                argument-name argument-value;
            }
            destination destination-name {
                retry-count number retry-interval seconds;
                transfer-delay seconds;
            }
            output-filename filename;
```

```
        output-format (text | xml);
        user-name name;
      }
      execute-commands {
        commands {
          "command";
        }
        destination destination-name {
          retry-count number retry-interval seconds;
          transfer-delay seconds;
        }
        output-filename filename;
        output-format (text | xml);
        user-name username;
      }
      ignore;
      raise-trap;
      upload filename (filename | committed) destination destination-name {
        retry-count number retry-interval seconds;
        transfer-delay seconds;
        user-name username;
      }
    }
  }
  traceoptions {
    file <filename> <files number> <match regular-expression> <size size> <world-readable |
      no-world-readable>;
    flag flag;
    no-remote-trace;
  }
```

This chapter discusses the following topics:

- Configuring Event Policies to Raise SNMP Traps on page 390
- Tracing Event Policy Processing on page 391

Using Correlated Events to Trigger an Event Policy

You can configure a policy that correlates two or more events. If the correlated events occur as specified, they cause particular actions to be taken. For example, you might want to issue certain operational mode commands when a **UI_CONFIGURATION_ERROR** event is generated within five minutes (300 seconds) after a **UI_COMMIT_PROGRESS** event. As another example, you might want to upload a particular file if a **DCD_INTERFACE_DOWN** event is generated two times within a 60-second interval.

To configure a policy that correlates events, include the following statements at the **[edit event-options]** hierarchy level:

```
[edit event-options]
policy policy-name {
   attributes-match {
      event1.attribute-name equals event2.attribute-name;
      event.attribute-name matches regular-expression;
      event1.attribute-name starts-with event2.attribute-name;
   }
   events [ events ];
   within seconds {
      events [ events ];
      not events [ events ];
      trigger (on | after | until) event-count;
   }
}
```

In the **events** statement, you can list multiple events. To view a list of the events that can be referenced in an event policy, issue the **set event-options policy *policy-name* events ?** configuration mode command:

```
user@host# set event-options policy policy-name events ?
Possible completions:
  <event>
  [               Open a set of values
  acct_accounting_ferror
  acct_accounting_fopen_error
  ...
```

Some of the system log messages that you can reference in an event policy are not listed in the output of the **set event-options policy *policy-name* events ?** command. For information about referencing these system log messages in your event policies, see “Using Nonstandard System Log Messages to Trigger Event Policies” on page 376.

In addition, you can reference internally generated events, which are discussed in “Generating Internal Events to Trigger Event Policies” on page 375.

The actions configured in the **then** statement are executed only if certain conditions are met, which you specify in the **within** and **attributes-match** statements.

You can configure a policy that is executed only if a specified event occurs within a specified time interval after another event. You do this by including the **within *seconds* events** statement. The policy is executed only if one or more of the events in the first **events** statement occur within a configured number of seconds after one or more of the events in the **within *seconds* events** statement. The number of seconds can be from 60 through 604,800. The **not** statement causes the policy to be executed only if the events do not occur within the configured time interval.

For example, the following policy is executed if ***event3***, ***event4***, or ***event5*** occurs within 60 seconds after ***event1*** or ***event2*** occurs:

```
[edit event-options]
policy 1 {
  events [ event3 event4 event5 ];
  within 60 events [ event1 event2 ];
  then {
    ...
  }
}
```

The **attributes-match** statement correlates two events as follows:

- ***event1.attribute-name* equals *event2.attribute-name***—Execute the policy only if the specified attribute of ***event1*** equals the specified attribute of ***event2***.
- ***event.attribute-name* matches *regular-expression***—Execute the policy only if the specified attribute of ***event*** matches a regular expression. For more information, see "Using Regular Expressions to Refine the Set of Events That Trigger a Policy" on page 374.
- ***event1.attribute-name* starts-with *event2.attribute-name***—Execute the policy only if the specified attribute of ***event1*** starts with the specified attribute of ***event2***.

You can include the **attributes-match** statement only if you include one or more **within** statements in the same policy configuration. This means the events are correlated only if they occur within a specified time period.

To view a list of all event attributes that you can reference, issue the **help syslog *event*** operational mode command. The output of this command shows the event attributes in angle brackets (<>). The following output shows that three attributes can be referenced for the **ACCT_ACCOUNTING_SMALL_FILE_SIZE** event: **filename**, **file-size**, and **record-size**.

```
user@host> help syslog ACCT_ACCOUNTING_SMALL_FILE_SIZE
Name:           ACCT_ACCOUNTING_SMALL_FILE_SIZE
Message:       File <filename> size (<file-size>) is smaller than record size
(<record-size>)
```

You can filter the output of a search by using the pipe (|) symbol. The following example lists the filters that can be used with the pipe symbol:

```
user@host> help syslog | ?
Possible completions:
  count                Count occurrences
  display              Show additional kinds of information
  except               Show only text that does not match a pattern
  find                 Search for first occurrence of pattern
  hold                 Hold text without exiting the --More-- prompt
```



```
last                Display end of output only
match               Show only text that matches a pattern
no-more             Don't paginate output
request             Make system-level requests
resolve             Resolve IP addresses
save                Save output text to file
trim                Trim specified number of columns from start of line
```

For more information about using the pipe symbol, see the *Junos OS CLI User Guide*.

Another way to view the attributes you can reference is by issuing the **set attributes-match *event*?** command at the **[edit event-options policy *policy-name*]** hierarchy level, as shown in the following example:

```
[edit event-options policy p1]
user@host# set attributes-match acct_accounting_small_file_size?
Possible completions:
  <from-event-attribute>  First attribute to compare
  acct_accounting_small_file_size.filename
  acct_accounting_small_file_size.filesize
  acct_accounting_small_file_size.record-size
```

NOTE: In this set command, there is no space between the event name and the question mark (?).

For configuration examples, see "Example: Ignoring Events Based on Receipt of Other Events" on page 402, "Example: Correlating Events Based on Event Attributes" on page 403, and "Controlling Event Policy Using a Regular Expression" on page 403.

Representing the Correlating Event in an Event Policy

As described in "Configuring an Event Policy to Execute Operational Mode Commands" on page 381, the double dollar sign (**$$**) notation represents the event that is triggering a policy. Triggering events are those that you configure at the **[edit event-options policy *policy-name* events]** hierarchy level.

As described in "Using Correlated Events to Trigger an Event Policy" on page 371, you can configure a policy that is executed only if a specified event occurs within a specified time interval after another event. You do this by including the **within *seconds* events** statement at the **[edit event-options policy *policy-name*]** hierarchy level:

```
[edit event-options policy policy-name ]
events [ events ];
within seconds events [ events ];
```

The policy is executed only if one or more of the events at the **[edit event-options policy *policy-name* events]** hierarchy level occur within a configured number of seconds after one or more of the events in the **within *seconds* events** statement.

For correlating events, the single dollar sign with the event name (**$*event***) notation represents the most recent event that matches the event name. The dollar sign with the asterisk (**$***) notation represents the most recent event that matches any of the correlating events.

For a configuration example, see “Example: Representing the Correlating Event in an Event Policy” on page 398.

Triggering an Event Policy Based on Event Count

You can configure an event policy to be triggered if an event or set of events occurs a specified number of times within a specified time period.

To do this, include the optional **trigger** statement at the **[edit event-options policy *policy-name* within *seconds*]** hierarchy level:

```
[edit event-options policy policy-name within seconds]
trigger (after | on | until) event-count;
```

The software counts the number of times the triggering event occurs. A triggering event can be any event configured at the **[edit event-options policy *policy-name* events]** hierarchy level. You can configure the following options:

- **after *event-count***—The policy is executed when the number of matching events received equals ***event-count*** plus one.
- **on *event-count***—The policy is executed when the number of matching events received equals ***event-count***.
- **until *event-count***—The policy is executed each time a matching event is received and stops being executed when the number of matching events received equals ***event-count***.

For configuration examples, see “Examples: Triggering a Policy Based on Event Count” on page 400.

Using Regular Expressions to Refine the Set of Events That Trigger a Policy

You can use regular expression matching to specify more exactly which events cause a policy to be executed.

To specify the text string that must appear in an event attribute for the policy to be executed, include the **matches** statement at the **[edit event-options policy *policy-name* attributes-match]** hierarchy level, and specify the regular expression which the event attribute must match:

```
[edit event-options policy policy-name attributes-match]
event.attribute-name matches regular-expression;
```

When you specify the regular expression, use the notation defined in POSIX Standard 1003.2 for extended (modern) UNIX regular expressions. Explaining regular expression syntax is beyond the scope of this document. Table 21 on page 375 specifies which character or characters are matched by some of the regular expression operators that you can use in the **matches** statement. In the descriptions, the term *term* refers to either a single alphanumeric character or a set of characters enclosed in square brackets, parentheses, or braces.

NOTE: The matches statement is not case-sensitive.

Table 21: Regular Expression Operators for the matches Statement

Operator	Matches
. (period)	One instance of any character except the space.
* (asterisk)	Zero or more instances of the immediately preceding term.
+ (plus sign)	One or more instances of the immediately preceding term.
? (question mark)	Zero or one instance of the immediately preceding term.
| (pipe)	One of the terms that appear on either side of the pipe operator.
! (exclamation point)	Any string except the one specified by the expression, when the exclamation point appears at the start of the expression. Use of the exclamation point is specific to the Junos OS.
^ (caret)	The start of a line, when the caret appears outside square brackets. One instance of any character that does not follow it within square brackets, when the caret is the first character inside square brackets.
$ (dollar sign)	The end of a line.
[ ] (paired square brackets)	One instance of one of the enclosed alphanumeric characters. To indicate a range of characters, use a hyphen (-) to separate the beginning and ending characters of the range. For example, **[a-z0-9]** matches any letter or number.
( ) (paired parentheses)	One instance of the evaluated value of the enclosed term. Parentheses are used to indicate the order of evaluation in the regular expression.

For a configuration example, see “Controlling Event Policy Using a Regular Expression” on page 403.

Generating Internal Events to Trigger Event Policies

Internal events are events you create yourself to trigger a policy to be executed. They are not generated by Junos OS processes, and they do not have any associated system log messages. You can generate an internal event based on a time interval or the time of day.

To generate an event, include the following statements at the **[edit event-options]** hierarchy level:

```
[edit event-options]
generate-event event-name {
    time-interval seconds;
    time-of-day hh:mm:ss;
}
```

In the **time-interval** statement, configure a frequency, in seconds, with which to repeatedly generate an event. The time interval can be from 60 through 2,592,000 seconds.

In the **time-of-day** statement, configure a time of day for the event to occur. Use the format ***hh*:*mm*:*ss***.

NOTE: If you modify the system time by issuing the **set date** operational mode command, we recommend that you also issue the **commit full** or the **restart event-process** command. Otherwise, an internal event based on the time of day might not be generated at the configured time.

For example, if you configure an internal event to be generated at 15:55:00, and then you modify the system time from 15:47:17 to 15:53:00, the event is generated when the system time is approximately 16:00 instead of at the configured time, 15:55:00. You can correct this problem by issuing the **commit full** or the **restart event-process** command.

For configuration examples, see “Example: Generating an Internal Event Every Hour” on page 404 and “Example: Generating an Internal Event at Midnight” on page 404.

Using Nonstandard System Log Messages to Trigger Event Policies

Some of the system log messages that you can reference in an event policy are not listed in the output of the **set event-options policy *policy-name* events ?** command. These system log messages have an event ID and a **message** attribute. Event IDs are based on the origin of the message, as shown in Table 22 on page 376.

Table 22: Event ID by System Log Message Origin

Event IDs	Origin
SYSTEM	Messages from UNIX domain sockets
KERNEL	Messages from the kernel
PIC	Messages from PICs
PFE	Messages from the Packet Forwarding Engine
LCC	On a TX Matrix router, messages from as line-card chassis (LCC)
SCC	On a TX Matrix router, messages from a switch-card chassis (SCC)

To base your event policy on the event types shown in Table 22 on page 376, include the **events *event-id*** statement and the **attributes-match** statement with the ***event-id*.message matches "*message*"** attribute at the **[edit event-options policy *policy-name*]** hierarchy level:

```
[edit event-options policy policy-name]
events event-id;
```

```
attributes-match {
  event-id.message matches "message";
}
```

For a configuration example, see “Example: Using Nonstandard System Log Messages to Trigger an Event Policy” on page 405.

Defining Destinations for File Archiving by Event Policies

When the action in an event policy generates output files, you might want to save them for later analysis. Similarly, you might want to save system files (such as system log or core files) from the time an event occurs. You must configure one or more *destinations* to which files can be uploaded for archiving. To define destinations, include the **destinations** statement at the **[edit event-options]** hierarchy level:

```
[edit event-options]
destinations {
  destination-name {
    archive-sites {
      url <password password>;
    }
    transfer-delay seconds;
  }
}
```

You can then reference configured destinations in an event policy. For information about referencing destinations, see “Configuring an Event Policy to Upload Files” on page 377 and “Configuring an Event Policy to Execute Operational Mode Commands” on page 381.

The optional **transfer-delay** statement allows you to specify the number of seconds the event process (eventd) waits before beginning to upload a file or multiple files. A transfer delay allows you to make sure a large file, such as a core file, is completely generated before the upload begins. For more information, see “Configuring the Delay Before Files Are Uploaded by an Event Policy” on page 379.

In the **archive-sites** statement, you can specify a destination as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification. URLs of the type file:// are not supported; however, local device directories are supported (for example, /var/tmp/). When you specify the archive site, do not add a forward slash (/) to the end of the URL. The format for the destination filename is *device-name_filename_YYYYMMDD_HHMMSS*.

Optionally, you can specify a plain-text password for login into an archive site.

For a configuration example, see “Example: Correlating Events Based on Receipt of Other Events Within a Specified Time Interval” on page 395.

Configuring an Event Policy to Upload Files

Various types of files are useful in diagnosing an event. These files include system log files, core files, and configuration files. When an event occurs, you can upload relevant files to a specified location for analysis.

To configure a policy that uploads files, include the following statements at the **[edit event-options]** hierarchy level:

```
[edit event-options]
policy policy-name {
   events [ events ];
   then {
      upload filename (filename | committed) destination destination-name {
         retry-count number retry-interval seconds;
         transfer-delay seconds;
         user-name username;
      }
   }
}
```

When an event policy uploads files for analysis, the files are named and time-stamped in the following format to ensure unique filenames:

device-name_filename_YYYYMMDD_HHMMSS

If a policy uploads multiple files within a 1-second period, the software gives each file a unique number as well, as follows:

device-name_filename_YYYYMMDD_HHMMSS_number

The number can be from 001 through 999. For example, if you have an event policy with output filename **rpd-messages** on **device1**, and this event policy is executed three times in 1 second, the files are named as follows:

- device1_rpd-messages_20070623_132333
- device1_rpd-messages_20070623_132333_001
- device1_rpd-messages_20070623_132333_002

In the **events** statement, you can list multiple events. If one or more of the listed events occurs, the upload action is executed. To view a partial list of the events that can be referenced in an event policy, issue the **set event-options policy *policy-name* events ?** configuration mode command:

```
[edit]
user@host# set event-options policy policy-name events ?
Possible completions:
 <event>
 [               Open a set of values
 acct_accounting_ferror
 acct_accounting_fopen_error
 ...
```

Some of the system log messages that you can reference in an event policy are not listed in the output of the **set event-options policy *policy-name* events ?** command. For information about referencing these system log messages in your event policies, see “Using Nonstandard System Log Messages to Trigger Event Policies” on page 376.

In addition, you can reference internally generated events, which are discussed in “Generating Internal Events to Trigger Event Policies” on page 375.

If desired, you can include multiple **upload** statements, one for each type of file to be archived. In the **filename** statement, specify a file or multiple files to be uploaded. You can specify multiple files with one **filename** configuration statement (sometimes called *filename globbing*). For example, to upload all files that are located in the **/var/log** directory and that start with the **messages** string, include the following statement:

```
upload filename /var/log/messages*;
```

To upload the committed configuration file, include the **upload filename committed destination** ***destination-name*** statement at the **[edit event-options policy** ***policy-name*** **then]** hierarchy level:

```
[edit event-options policy policy-name then]
upload filename committed destination destination-name;
```

For the **destination** statement, specify a destination name that you configured at the **[edit event-options destinations]** hierarchy level. For more information, see "Defining Destinations for File Archiving by Event Policies" on page 377.

Configuring the Delay Before Files Are Uploaded by an Event Policy

A transfer delay allows you to specify the number of seconds the event process (eventd) waits before beginning to upload a file or multiple files. A transfer delay allows you to ensure that a large file, such as a core file, is completely generated before the upload begins.

As described in "Defining Destinations for File Archiving by Event Policies" on page 377, you can associate a transfer delay with a destination. If you associate a transfer delay with a destination, the transfer delay applies to all file upload actions that use the destination.

In the following example, the ***some-dest*** destination is common for both event policies, **policy1** and **policy2**. A transfer delay of 2 seconds is associated with the **some-dest** destination and applies to uploading the output files to the destination for both event policies.

```
[edit event-options]
policy policy1 {
    events e1;
    then {
    execute-commands {
        commands {
            "show version";
        }
        output-filename command-output.txt;
        destination some-dest;
    }
}
policy policy2 {
    events e2;
    then {
        event-script bar.xsl {
            output-filename event-script-output.txt;
            destination some-dest;
        }
```

```
    }
  }
  destinations {
    some-dest {
      transfer-delay 2;
      archive-sites {
        "http://robot@my.big.com/foo/moo" password "password";
        "http://robot@my.little.com/foo/moo" password "password";
      }
    }
  }
```

Suppose you have multiple event policy actions that use the same destination. For some of these event policy actions, you want a transfer delay, and for other event policy actions you want no transfer delay. To assign a transfer delay to a single event policy action, include the optional **transfer-delay** statement for each action:

transfer-delay *seconds*;

You can include this statement at the following hierarchy levels:

- **[edit event-options policy** *policy-name* **then event-script** *filename* **destination** *destination-name***]**
- **[edit event-options policy** *policy-name* **then execute-commands destination** *destination-name***]**
- **[edit event-options policy** *policy-name* **then upload filename (***filename* **| committed) destination** *destination-name***]**

If you configure a transfer delay at the **[edit event-options destinations** *destination-name***]** hierarchy level, and you also configure a transfer delay for the event policy action, the resulting transfer delay is the sum of the two:

```
Total transfer-delay =
transfer-delay (destination) + transfer-delay (event-policy-action)
```

For configuration examples, see “Examples: Assigning a Transfer Delay to an Event Policy Action” on page 396.

Configuring an Event Policy to Retry the File Upload Action

Transient network problems can cause a file upload operation to fail. When this happens, you might want to retry the file upload operation. By default, if the file upload operation fails for any reason, the event policy does not retry the upload operation.

To configure the policy to retry a file upload operation, include the optional **retry-count** and **retry-interval** statements:

retry-count *number* retry-interval *seconds*;

You can include these statements at the following hierarchy levels:

- **[edit event-options policy** *policy-name* **then event-script** *filename* **destination** *destination-name***]**

- **[edit event-options policy** *policy-name* **then execute-commands destination** *destination-name***]**
- **[edit event-options policy** *policy-name* **then upload filename (***filename* **| committed) destination** *destination-name***]**

The **retry-count** statement sets the number of times the policy retries the upload operation if the upload fails. The default value for the **retry-count** statement is 0 and the maximum is 10.

If you include the **retry-count** statement, you can also include the **retry-interval** statement, which sets the time interval (in seconds) between each retry.

For configuration examples, see "Examples: Retrying the File Upload Action" on page 399.

Configuring an Event Policy to Execute Operational Mode Commands

Operational mode commands request that the device running Junos OS perform an operation or provide diagnostic output. They allow you to view statistics and information about a device's current operating status. They also allow you to take corrective actions, such as restarting software processes, taking a PIC offline and back online, switching to redundant interfaces, and adjusting Label Switching Protocol (LSP) bandwidth. For more information about operational mode commands, see the following references:

- *Junos OS Interfaces Command Reference*
- *Junos OS Routing Protocols and Policies Command Reference*
- *Junos OS System Basics and Services Command Reference*

You can configure a policy that causes operational mode commands to be issued and the output of those commands to be uploaded to a specified location for analysis.

To configure such a policy, include the following statements at the **[edit event-options]** hierarchy level:

```
[edit event-options]
policy policy-name {
   events [ events ];
   then {
      execute-commands {
         commands {
            "command";
         }
         output-filename filename;
         output-format (text | xml);
         destination destination-name;
      }
   }
}
```

In the **events** statement, you can list multiple events. If one or more of the listed events occurs, the operational mode commands are issued. To view a list of the events that can be referenced in an event policy, issue the **set event-options policy *policy-name* events ?** configuration mode command:

```
[edit]
user@host# set event-options policy policy-name events ?
Possible completions:
 <event>
 [               Open a set of values
 acct_accounting_ferror
 acct_accounting_fopen_error
 ...
```

Some of the system log messages that you can reference in an event policy are not listed in the output of the **set event-options policy *policy-name* events ?** command. For information about referencing these system log messages in your event policies, see "Using Nonstandard System Log Messages to Trigger Event Policies" on page 376.

In addition, you can reference internally generated events, which are discussed in "Generating Internal Events to Trigger Event Policies" on page 375.

In the **commands** statement, you can issue multiple operational mode commands upon receipt of a specific event. Enclose each command in quotation marks (**" "**). The eventd process issues the commands in the order in which they appear in the configuration. For example, in the following configuration, the execution of **policy1** causes the **show interfaces** command to be issued first, followed by the **show chassis alarms** command:

```
[edit event-options policy policy1 then execute-commands]
user@host# show
commands {
   "show interfaces";
   "show chassis alarms";
}
```

You can include variables in the command to allow data from the triggering event to be automatically included in the command syntax. The eventd process replaces each variable with values contained in the event that triggers the policy. You can use command variables of the following forms:

- **{$$.*attribute-name*}**—The double dollar sign (**$$**) notation represents the event that is triggering a policy. When combined with an attribute name, the variable is replaced by the value of the attribute name in the triggering event. For example, **{$$.interface-name}** stands for the value of the **interface-name** attribute in the triggering event.
- **{$*event*.*attribute-name*}**—The **{$*event*.*attribute-name*}** notation represents the most recent event that matches the specified event. The variable is replaced by the value of the attribute name of the most recent event that matches ***event***. For example, when a policy issues the **show interfaces {$COSD_CHAS_SCHED_MAP_INVALID.interface-name}** command, the **{$COSD_CHAS_SCHED_MAP_INVALID.interface-name}** variable is substituted by the **interface-name** attribute of the most recent **COSD_CHAS_SCHED_MAP_INVALID** event cached by the event process.

For a given event, you can view a list of event attributes that you can reference in an operational mode command by issuing the **help syslog *event-name*** command:

```
user@host> help syslog event-name
```

For example, in the following command output, text in angle brackets (< >) shows that **classifier-type** is an attribute of the **cosd_unknown_classifier** event:

```
user@host> help syslog cosd_unknown_classifier
Name:          COSD_UNKNOWN_CLASSIFIER
Message:       rtsock classifier type <classifier-type> is invalid
...
```

You can filter the output of a search by using the pipe (**|**) symbol. The following example lists the filters that can be used with the pipe symbol:

```
user@host# help syslog | ?
Possible completions:
  count                Count occurrences
  display              Show additional kinds of information
  except               Show only text that does not match a pattern
  find                 Search for first occurrence of pattern
  hold                 Hold text without exiting the --More-- prompt
  last                 Display end of output only
  match                Show only text that matches a pattern
  no-more              Don't paginate output
  request              Make system-level requests
  resolve              Resolve IP addresses
  save                 Save output text to file
  trim                 Trim specified number of columns from start of line
```

For more information about using the pipe symbol, see the *Junos OS CLI User Guide*.

Another way to view a list of event attributes is to issue the **set attributes-match *event*?** configuration mode command at the **[edit event-options policy *policy-name*]** hierarchy level:

```
[edit event-options policy policy-name]
user@host# set attributes-match event ?
```

For example, in the following command output, the ***event.attribute*** list shows that **classifier-type** is an attribute of the **cosd_unknown_classifier** event:

```
[edit event-options policy policy-name]
user@host# set attributes-match cosd_unknown_classifier?
Possible completions:
 <from-event-attribute>  First attribute to compare
 cosd_unknown_classifier.classifier-type
```

NOTE: In this set command, there is no space between the event name and the question mark (?).

To view a list of all events that you can reference, issue the **set attributes-match ?** configuration mode command at the **[edit event-options policy *policy-name*]** hierarchy level:

```
[edit event-options policy policy-name]
user@host# set attributes-match ?
Possible completions:
  <from-event-attribute>  First attribute to compare
  acct_accounting_ferror
  acct_accounting_fopen_error
  ...
```

In the **output-filename** statement, assign the name of the file to which to write command output for the specified commands. The filename format is *hostname_filename_YYYYMMDD_HHMMSS_index-number*.

For each uploaded file, a hostname and timestamp ensure that the uploaded files have unique filenames. If a policy is triggered multiple times in a 1-second period, an index number is added to ensure the filenames are unique. The index number range is 001 through 999.

For example, on a device named **r1**, if you configure the output filename to be **ifl-events**, and this event policy is triggered three times in 1 second, the files are named:

- r1_ifl-events_20060623_132333
- r1_ifl-events_20060623_132333_001
- r1_ifl-events_20060623_132333_002

By default, the command output format is Junos Extensible Markup Language (XML). To change this, include the **output-format text** statement. This causes the command output to be in formatted ASCII text.

In the **destination** statement, include the destination name that you configured at the **[edit event-options destinations]** hierarchy level. For more information, see “Defining Destinations for File Archiving by Event Policies” on page 377.

For a configuration example, see “Example: Correlating Events Based on Receipt of Other Events Within a Specified Time Interval” on page 395.

Executing Event Scripts in an Event Policy

Event scripts are Extensible Stylesheet Transformation (XSLT) or Stylesheet Language Alternative Syntax (SLAX) scripts that you write and that are run when triggered by an event policy. Event scripts can perform any function available through Junos XML or Junos XML protocol remote procedure calls (RPCs). Additionally, you can pass to an event script a set of arguments that you define.

A script can change the device configuration, build and run an operational mode command, receive the command output, inspect the output, and determine the next appropriate action. This process can be repeated until the source of the problem is determined. The script can then report the source of the problem to you on the CLI.

You can configure an event policy that causes event scripts to be run and the output of those scripts to be uploaded to a specified location for analysis.

To configure such a policy, include the following statements at the **[edit event-options]** hierarchy level:

```
[edit event-options]
policy policy-name {
    events [ events ];
    then {
        event-script filename {
            arguments {
                argument-name argument-value;
            }
            output-filename filename;
            output-format (text | xml);
            destination destination-name;
        }
    }
}
```

In the **events** statement, you can list multiple events. If one or more of the listed events occurs, the event script is executed. To view a list of the events that can be referenced in an event policy, issue the **set event-options policy *policy-name* events ?** configuration mode command:

```
[edit]
user@host# set event-options policy policy-name events ?
Possible completions:
  <event>
  [             Open a set of values
  acct_accounting_ferror
  acct_accounting_fopen_error
  ...
```

Some of the system log messages that you can reference in an event policy are not listed in the output of the **set event-options policy *policy-name* events ?** command. For information about referencing these system log messages in your event policies, see "Using Nonstandard System Log Messages to Trigger Event Policies" on page 376.

In addition, you can reference internally generated events, which are discussed in "Generating Internal Events to Trigger Event Policies" on page 375.

In the **event-script** statement, you can specify a script to be executed on receipt of an event. The eventd process runs the scripts in the order in which they appear in the configuration. The scripts that you reference in the **event-script** statement must be located in the /var/db/scripts/event directory on the device's hard drive or the /config/scripts/event directory on the flash drive. For more information about event script file location, see "How Event Scripts Work" on page 437. Furthermore, the event scripts must be enabled at the **[edit event-options event-script file]** hierarchy level. For more information, see "Installing Event Scripts on a Router" on page 448 and "Event Policy Examples" on page 395.

You can include arguments to the script as name/value pairs. You can include variables in the argument values to allow data from the triggering event to be automatically included in the argument. The eventd process replaces each variable with values contained in the event that triggers the policy. You can use variables of the following forms:

- **{$$.*attribute-name*}**—The double dollar sign ($$) notation represents the event that is triggering a policy. When combined with an attribute name, the variable is replaced by the value of the attribute name in the triggering event. For example, **{$$.interface-name}** stands for the value of the **interface-name** attribute in the triggering event.
- **{$*event*.*attribute-name*}**—The **{$*event*.*attribute-name*}** notation represents the most recent event that matches the specified event. The variable is replaced by the value of the attribute name of the most recent event that matches ***event***. For example, when you include an argument called **interface** and define the value as **{$COSD_CHAS_SCHED_MAP_INVALID.interface-name},** the **{$COSD_CHAS_SCHED_MAP_INVALID.interface-name}** variable is replaced by the **interface-name** attribute of the most recent **COSD_CHAS_SCHED_MAP_INVALID** event cached by the eventd process.

For a given event, you can view a list of event attributes that you can reference by issuing the **help syslog *event*** command:

```
user@host> help syslog event-name
```

For example, in the following command output, text in angle brackets (**< >**) shows attributes of the **COSD_CHASSIS_SCHEDULER_MAP_INVALID** event:

```
user@host> help syslog COSD_CHASSIS_SCHEDULER_MAP_INVALID
Name:        COSD_CHASSIS_SCHEDULER_MAP_INVALID
Message:     Chassis scheduler map incorrectly applied to interface
<interface-name>: <error-message>
...
```

You can filter the output of a search by using the pipe (**|**) symbol. The following example lists the filters that can be used with the pipe symbol:

```
user@host> help syslog | ?
Possible completions:
 count               Count occurrences
 display             Show additional kinds of information
 except              Show only text that does not match a pattern
 find                Search for first occurrence of pattern
 hold                Hold text without exiting the --More-- prompt
 last                Display end of output only
 match               Show only text that matches a pattern
 no-more             Don't paginate output
 request             Make system-level requests
 resolve             Resolve IP addresses
 save                Save output text to file
 trim                Trim specified number of columns from start of line
```

For more information about using the pipe symbol, see the *Junos OS CLI User Guide*.

Another way to view a list of event attributes is to issue the **set attributes-match *event* ?** configuration mode command at the **[edit event-options policy *policy-name*]** hierarchy level:

```
[edit event-options policy policy-name]
user@host# set attributes-match event ?
```

For example, in the following command output, the ***event.attribute*** list shows that **error-message** and **interface-name are** attributes of the **cosd_chassis_scheduler_map_invalid** event:

```
[edit event-options policy p1]
user@host# set attributes-match cosd_chassis_scheduler_map_invalid?
Possible completions:
 <from-event-attribute>  First attribute to compare
 cosd_chassis_scheduler_map_invalid.error-message
 cosd_chassis_scheduler_map_invalid.interface-name
```

In this **set** command, there is no space between the event name and the question mark (**?**).

To view a list of all event attributes that you can reference, issue the **set attributes-match ?** configuration mode command at the **[edit event-options policy *policy-name*]** hierarchy level:

```
[edit event-options policy policy-name]
user@host# set attributes-match ?
Possible completions:
 <from-event-attribute>  First attribute to compare
 acct_accounting_ferror
 acct_accounting_fopen_error
 ...
```

By default, the command output format is text. To change this, include the **output-format xml** statement.

In the optional **output-filename** statement, assign the name of the file to which to write script output for the specified script.

The filename format is ***hostname_filename_YYYYMMDD_HHMMSS_index-number***.

For each uploaded file, a hostname and timestamp are automatically added to the filename to ensure that the uploaded files have unique filenames. If a policy is triggered multiple times in a 1-second period, an index number is added to ensure the filenames are unique. The index number range is 001 through 999.

For example, on a device named **r1**, if you configure the output filename to be **ifl-events**, and this event policy is triggered three times in 1 second, the files are named:

- r1_ifl-events_20060623_132333
- r1_ifl-events_20060623_132333_001
- r1_ifl-events_20060623_132333_002

In the optional **destination** statement, include the destination name that you configured at the **[edit event-options destinations]** hierarchy level. For more information, see “Defining Destinations for File Archiving by Event Policies” on page 377.

For the **output-filename** and **destination** statements, there are four configuration scenarios:

- You can omit the **output-filename** and **destination** statements. This option makes sense when the event script has no output. For example, the event script might execute only **request** commands, which have no output.
- You can include the **destination** statement in the configuration. You omit the **output-filename** statement in the configuration and specify an output filename in the event script instead. The script output is sent to the destination specified in the configuration. If you do not include the **destination** statement in the configuration, the script output is not uploaded.

  In this scenario, the event policy extracts the filename from the event script. The event script writes the output filename as **STDOUT**. The XML syntax to use in the event script is:

  ```
  <output>
    <event-script-output-filename>filename</event-script-output-filename>
  </output>
  ```

  The **<event-script-output-filename>** element must be the first child tag within the **<output>** parent tag.

  On a device named **device2**, configure an event script action with a destination **host**, and omit the **output-filename** statement. Define the destination **host** as ftp://user@device1//tmp.

  In the **script1.xsl** event script, write the following output to **STDOUT**:

  ```
  <event-script-output-filename>/var/cmd.txt</event-script-output-filename>
  ```

  Configure the **policy1** event policy as follows:

  ```
  [edit event-options]
  policy policy1 {
    then {
      event-script script1.xsl {
        destination host;
      }
    }
  }
  destinations {
    host {
      archive-sites {
        "ftp://user@device1//tmp" password "$9$xkJNbYg4ZDH.oJ.fQnpuSyl";  ##
          SECRET-DATA***
      }
    }
  }
  ```

  In this example, the /var/cmd.txt file resides on device **device2**. The event policy uses the File Transfer Protocol (FTP) to upload this file to the /tmp directory on device **device1**.

  The event policy reads the output filename /var/cmd.txt from **STDOUT**. Then the event policy uploads the /var/cmd.txt file to the configured destination, which is the /tmp directory on device **device1**. The event policy renames the /var/cmd.txt file as device2_cmd.txt_*YYYYMMDD_HHMMSS_range*.

- You can include the **output-filename** and **destination** statements. If you include the **output-filename** statement in the configuration, you must also include the **destination** statement in the configuration. In this case, the script output is redirected to the output filename specified in the configuration and is sent to the destination specified in the configuration.
- You can include the **output-filename** and **destination** statements, and also specify an output filename directly within the event script. If you do this, the output filename specified in the configuration overrides the output filename specified in the event script.

Configuring Event Policies to Ignore an Event

You can modify a policy to cause particular events to be ignored or to cause all events to be ignored during a particular time interval, to allow for maintenance for example. To configure such a policy, include the following statements at the **[edit event-options]** hierarchy level:

```
[edit event-options]
policy policy-name {
   events [ events ];
   then {
      ignore;
   }
}
```

In the **events** statement, you can list multiple events. To view a list of the events that can be referenced in an event policy, issue the **set event-options policy *policy-name* events ?** configuration mode command:

```
[edit]
user@host# set event-options policy policy-name events ?
Possible completions:
 <event>
 [              Open a set of values
 acct_accounting_ferror
 acct_accounting_fopen_error
 ...
```

Some of the system log messages that you can reference in an event policy are not listed in the output of the **set event-options policy *policy-name* events ?** command. For information about referencing these system log messages in your event policies, see “Using Nonstandard System Log Messages to Trigger Event Policies” on page 376.

In addition, you can reference internally generated events, which are discussed in “Generating Internal Events to Trigger Event Policies” on page 375.

If one or more of the listed events occur, a system log message for the event is not generated, and no further policies associated with this event are processed. If you include the **ignore** statement in a policy configuration, you cannot configure any other actions in the policy.

For configuration examples, see “Example: Ignoring Events Based on Receipt of Other Events” on page 402.

Changing the User Privilege Level for an Event Policy Action

Only superusers can configure event policies. Event policy actions—such as executing event scripts, uploading files, and executing operational mode commands—are by default executed by user **root**, because the event process (eventd) runs with **root** privileges.

In some cases, you might want an event policy action to be executed with restricted privileges. For example, suppose you configure an event policy that executes a script if an interface goes down. The script includes remote procedure calls (RPCs) to change the device configuration if certain conditions are present. If you do not want the script to change the configuration, you can execute the script with a restricted user profile. When the script is executed with a user profile that disallows configuration changes, the RPCs to change the configuration fail.

You can associate a user with each action in an event policy. If a user is not associated with an event policy action, then the action is executed as user **root** by default.

To specify the user under whose privileges an action is executed, include the **user-name** statement:

user-name *username*;

You can include this statement at the following hierarchy levels:

- **[edit event-options policy** *policy-name* **then event-script** *filename***]**
- **[edit event-options policy** *policy-name* **then execute-commands]**
- **[edit event-options policy** *policy-name* **then upload filename (***filename* **| committed) destination** *destination-name***]**

NOTE: The username that you specify must be configured at the **[edit system login]** hierarchy level. For more information, see the *Junos OS System Basics Configuration Guide*.

For a configuration example, see “Example: Associating an Optional User with an Event Policy Action” on page 398.

Configuring Event Policies to Raise SNMP Traps

SNMP *traps* enable an agent to notify a network management system (NMS) of significant events by way of an unsolicited SNMP message. You can configure an event policy action that raises traps for events based on system log messages. This enables notification of an SNMP trap-based application when an important system log message occurs. You can convert any system log message (for which there are no corresponding traps) into a trap. This is valuable if you use NMS traps rather than system log messages to monitor your network.

To configure a policy that raises a trap on receipt of an event, include the following statements at the **[edit event-options policy *policy-name*]** hierarchy level:

```
[edit event-options policy policy-name]
events [ events ];
then {
  raise-trap;
}
```

The MIB (**jnx-syslog.mib**) supports this policy action. For more information, see the *Junos OS SNMP MIBs and Traps Reference*.

For a configuration example, see "Example: Raising an SNMP Trap in Response to an Event" on page 405.

Tracing Event Policy Processing

Event policy tracing operations track all event policy operations and record them in a log file. The logged error descriptions provide detailed information to help you solve problems faster.

By default, no events are traced. If you include the **traceoptions** statement at the **[edit event-options]** hierarchy level, the default tracing behavior is the following:

- Important events are logged in a file called **eventd** located in the **/var/log** directory.
- When the file **eventd** reaches 128 kilobytes (KB), it is renamed and compressed to **eventd.0.gz**, then **eventd.1.gz**, and so on, until there are three trace files. Then the oldest trace file (**eventd.2.gz**) is overwritten. (For more information about how log files are created, see the *Junos OS System Log Messages Reference*.)
- Log files can be accessed only by the user who configures the tracing operation.

You cannot change the directory (**/var/log**) to which trace files are written. However, you can customize the other trace file settings by including the following statements at the **[edit event-options traceoptions]** hierarchy level:

```
[edit event-options traceoptions]
file <filename> <files number> <match regular-expression> <size size> <world-readable |
  no-world-readable>;
flag all;
flag configuration;
flag database;
flag events;
flag policy;
flag server;
flag syslog
flag timer-events;
no-remote-trace;
```

These statements are described in the following sections:

- Configuring the Event Policy Log Filename on page 392
- Configuring the Number and Size of Event Policy Log Files on page 392
- Configuring Access to the Log File on page 392

Configuring the Event Policy Log Filename

By default, the name of the file that records trace output is **eventd**. You can specify a different name by including the **file** statement at the **[edit event-options traceoptions]** hierarchy level:

```
[edit event-options traceoptions]
file filename;
```

Configuring the Number and Size of Event Policy Log Files

By default, when the trace file reaches 128 kilobytes (KB) in size, it is renamed ***filename*.0**, then ***filename*.1**, and so on, until there are three trace files. Then the oldest trace file (***filename*.2**) is overwritten.

You can configure the limits on the number and size of trace files by including the following statements at the **[edit event-options traceoptions file <*filename*>]** hierarchy level:

```
[edit event-options traceoptions file <filename>]
files number size size;
```

For example, set the maximum file size to 2 MB and the maximum number of files to 20. When the file that receives the output of the tracing operation (*filename*) reaches 2 MB, *filename* is renamed and compressed to *filename*.0.gz and a new file called *filename* is created.

When *filename* reaches 2 MB, *filename*.0.gz is renamed *filename*.1.gz and *filename* is renamed and compressed to *filename*.0.gz. This process repeats until there are 20 trace files. Then the oldest file (*filename*.19.gz) is overwritten.

The number of files can range from 2 through 1000 files. The file size can range from 10 KB through 1 gigabyte (GB).

Configuring Access to the Log File

By default, log files can be accessed only by the user who configures the tracing operation.

To specify that any user can read all log files, include the **world-readable** statement at the **[edit event-options traceoptions file <*filename*>]** hierarchy level:

```
[edit event-options traceoptions file <filename>]
world-readable;
```

To explicitly set the default behavior, include the **no-world-readable** statement at the **[edit event-options traceoptions file <*filename*>]** hierarchy level:

```
[edit event-options traceoptions file <filename>]
no-world-readable;
```

Configuring a Regular Expression for Lines to Be Logged

By default, the trace operation output includes all lines relevant to the logged events.

You can refine the output by including the **match** statement at the **[edit event-options traceoptions file <*filename*>]** hierarchy level and specifying a regular expression to be matched:

```
[edit event-options traceoptions file <filename>]
match regular-expression;
```

Configuring the Trace Operations

By default, only important events are logged. You can configure the trace operations to be logged by including the following statements at the **[edit event-options traceoptions]** hierarchy level:

```
[edit event-options traceoptions]
flag all;
flag configuration;
flag database;
flag events;
flag policy;
flag server;
flag syslog
flag timer-events;
```

Table 23 on page 393 describes the meaning of the event policy tracing flags.

Table 23: Event Policy Tracing Flags

Flag	Description	Default Setting
all	Trace all operations.	Off
configuration	Log reading of configuration at the **[edit event-options]** hierarchy level.	Off
events	Trace important events.	Off
database	Log events involving storage and retrieval in events database.	Off
policy	Log policy processing.	Off
server	Log communication with processes that are generating events.	Off
syslogd	Log syslog related traces	Off
timer-events	Log internally generated events.	Off

To display the end of the log, issue the **show log eventd | last** operational mode command:

```
[edit]
user@host# run show log eventd | last
```

CHAPTER 24

Event Policy Examples

This chapter includes the following examples:

Example: Correlating Events Based on Receipt of Other Events Within a Specified Time Interval

In the following policy, a set of commands is issued and the output is logged and saved to a given location. The policy is executed if ***event3***, ***event4***, or ***event5*** occurs within 60 seconds after ***event1*** or ***event2*** occurs. The pseudocode for the policy is as follows:

```
if this event is (event3 or event4 or event5)
  and
  (event1 or event2 has been received within the last 60 seconds)
then {
  run a set of commands;
  log the output of these commands to a location;
}
```

Specify two archive sites in the configuration. The device attempts to transfer to the first archive site in the list, moving to the next site only if the transfer fails.

```
[edit event-options]
policy 1 {
   events [ event3 event4 event5 ];
   within 60 events [ event1 event2 ];
   then {
      execute-commands {
         commands {
            "command";
         }
         output-filename my_cmd_out;
         destination policy-1-command-dest;
      }
   }
}
destinations {
   policy-1-command-dest {
      archive-sites {
         http://robot@my.big.com/a/b;
         http://robot@my.little.com/a/b;
      }
   }
}
```

Examples: Assigning a Transfer Delay to an Event Policy Action

This section discusses three examples.

Example 1 Configure two event policies, **policy1** and **policy2**. The **policy1** event policy has a 5-second transfer-delay when uploading the process.core file to the **some-dest** destination. The **policy2** event policy has no transfer delay when uploading the process.core file to the same destination.

```
[edit event-options]
policy policy1 {
   events e1;
   then {
      upload filename process.core destination some-dest {
         transfer-delay 5;
      }
   }
}
policy policy2 {
   events e2;
   then {
      upload filename process.core destination some-dest;
   }
}
destinations {
   some-dest {
      archive-sites {
         "http://robot@my.little.com/foo/moo" password "password";
         "http://robot@my.big.com/foo/moo" password "password";
      }
```

```
    }
  }
```

Example 2 The **policy1** event policy has a 7-second (5 seconds + 2 seconds) transfer delay when uploading the **process.core** file to the destination. The **policy2** event policy has a 2-second transfer delay when uploading the **process.core** file to the destination.

```
[edit event-options]
policy policy1 {
  events e1;
  then {
    upload filename process.core destination some-dest {
      transfer-delay 5;
    }
  }
}
policy policy2 {
  events e2;
  then {
    upload filename process.core destination some-dest;
  }
}
destinations {
  some-dest {
    transfer-delay 2;
    archive-sites {
      "http://robot@my.little.com/foo/moo" password "password";
      "http://robot@my.big.com/foo/moo" password "password";
    }
  }
}
```

Example 3 The **policy1** event-policy is executed with **user1** privileges and uploads the process.core file after a transfer delay of 7 seconds (5 seconds + 2 seconds). The **policy2** event policy is executed with **root** privileges and uploads the process.core file after a transfer delay of 6 seconds (4 seconds + 2 seconds).

```
[edit event-options]
policy policy1 {
  events e1;
  then {
    upload filename process.core destination some-dest {
      transfer-delay 5;
      user-name user1;
    }
  }
}
policy policy2 {
  events e2;
  then {
    upload filename process.core destination some-dest {
      transfer-delay 4;
    }
  }
}
destinations {
```

```
    some-dest {
      transfer-delay 2;
      archive-sites {
        "http://robot@my.little.com/foo/moo" password "password";
        "http://robot@my.big.com/foo/moo" password "password";
      }
    }
}
```

Example: Representing the Correlating Event in an Event Policy

```
[edit event-options]
policy p1 {
  events [ e1 e2 e3 ];
  within 60 events [ e4 e5 e6 ];
  then {
    execute-commands {
      commands {
        "show interfaces {$$.interface-name}";
        "show interfaces {$e4.interface-name}";
        "show interfaces {$*.interface-name}";
      }
      output-filename command-output.txt;
      destination some-dest;
    }
  }
}
```

In the **show interfaces {$$.interface-name}** command, the value of the **interface-name** attribute of event **e1**, **e2**, or **e3** is substituted for the **{$$.interface-name}** variable.

In the **show interfaces {$e4.interface-name}** command, the value of the **interface-name** attribute of the most recent **e4** event is substituted for the **{$e4.interface-name}** variable.

In the **show interfaces {$*.interface-name}** command, the value of the **interface-name** attribute of the most recent **e4**, **e5**, or **e6** event is substituted for the **{$*.interface-name}** variable. If one of **e4**, **e5**, or **e6** occurs within 60 seconds of **e1**, **e2**, or **e3**, the value of the **interface-name** attribute for that correlating event (**e4**, **e5**, or **e6**) is substituted for the **{$*.interface-name}** variable. If the correlating event does not have an **interface-name** attribute, the software does not execute the **show interfaces {$*.interface-name}** command.

If both **e4** and **e5** occur within 60 seconds of **e1**, then the value of the **interface-name** attribute for **e4** is substituted for the **{$*.interface-name}** variable. This is because the event process (eventd) searches for correlating events in sequential order as configured in the **within** statement. In this case, the order is **e4** > **e5** > **e6**.

Example: Associating an Optional User with an Event Policy Action

Configure two event policies, **policy1** and **policy2**.

In **policy1**, associate user **user1** with the **execute-commands** action. The **execute-commands** action is executed with **user1** privileges.

In **policy2**, do not explicitly associate a user with the **event-script** action. The **event-script** action is executed with **root** privileges.

```
[edit system]
login {
   user user1 {
      class operator;
   }
}
[edit event-options]
policy p1 {
   events e1;
   then {
      execute-commands {
         commands {
            "show version";
         }
         user-name user1;
         output-filename command-output.txt;
         destination some-dest;
      }
   }
}
policy p2 {
   events e2;
   then {
      event-script script.xsl {
         output-filename event-script-output.txt;
         destination some-dest;
      }
   }
}
```

Examples: Retrying the File Upload Action

This section discusses two examples.

Example 1 Configure a policy that retries the file upload operation two times with a time interval of 5 seconds between retries:

```
event-options {
   policy p1 {
      events e1;
      then {
         execute-commands {
            commands {
               command1;
            }
            output-filename command-output.txt;
            destination some-dest {
               retry-count 2 retry-interval 5;
            }
         }
      }
   }
}
```

Example 2 Configure a transfer delay of 10 seconds and retry the file upload operation two times with a time interval of 5 seconds between retries:

```
event-options {
  policy p2 {
    events e1;
    then {
      execute-commands {
        commands {
          command1;
        }
        output-filename command-output.txt;
        destination some-dest {
          retry-count 2 retry-interval 5;
          transfer-delay 10;
        }
      }
    }
  }
}
```

The transfer delay is in operation for the first upload attempt only. The policy uploads the command-output.txt file after a 10-second transfer delay. If the event process (eventd) detects failure of the upload operation, eventd retries the upload operation after 5 seconds. The failure detection time can be in the range from 60 to 90 seconds, depending on the transmission protocol, such as FTP.

The following sequence describes the file upload operation with two failed retransmissions:

1. Policy triggers upload operation.
2. Transmission delay of 10 seconds.
3. Policy tries to upload the output file.
4. Policy detects transmission failure.
5. Retry interval of 5 seconds.
6. Policy tries to upload the output file.
7. Policy detects transmission failure.
8. Retry interval of 5 seconds.
9. Policy tries to upload the output file.
10. Policy detects transmission failure.
11. Policy declares the failure of the file upload operation.

Examples: Triggering a Policy Based on Event Count

This section discusses two examples.

NOTE: The RADIUS_LOGIN_FAIL, TELNET_LOGIN_FAIL, and SSH_LOGIN_FAIL events are not actual Junos OS events. They are illustrative for these examples.

Example 1 Configure an event policy called **login**. The **login** policy is executed if five login failure events (**RADIUS_LOGIN_FAIL**, **TELNET_LOGIN_FAIL**, or **SSH_LOGIN_FAIL**) are generated within 120 seconds. Take action by executing the **login-fail.xsl** event script, which disables the user account.

```
[edit event-options]
policy login {
   events [ RADIUS_LOGIN_FAIL TELNET_LOGIN_FAIL SSH_LOGIN_FAIL ];
   within 120 {
      trigger after 4;
   }
   then {
      event-script login-fail.xsl {
         destination some-dest;
      }
   }
}
```

Table 24 on page 401 shows how events add to the count.

Table 24: Event Count Triggers Policy

Event Number	Event	Time	Count	Order
1	RADIUS_LOGIN_FAIL	00:00:00	1	[1]
2	TELNET_LOGIN_FAIL	00:00:20	2	[1 2]
3	RADIUS_LOGIN_FAIL	00:02:05	2	[2 3]
4	SSH_LOGIN_FAIL	00:02:40	2	[3 4]
5	TELNET_LOGIN_FAIL	00:02:55	3	[3 4 5]
6	TELNET_LOGIN_FAIL	00:03:01	4	[3 4 5 6]
7	RADIUS_LOGIN_FAIL	00:03:55	5	[3 4 5 6 7]

The columns in Table 24 on page 401 mean the following:

- Event number—Event sequence number.
- Event—Policy login events received by the event process (eventd).
- Time—Time (in ***hh*:*mm*:*ss*** format) when eventd receives the event.
- Count—The number of events received by eventd within the last 120 seconds.
- Order—Order of events as received by eventd within the last 120 seconds.

At time 00:03:55, the value of count is more than 4; therefore, the **login** policy executes the **login-fail.xsl** script.

Example 2 Configure an event policy called **login**. The **login** policy is executed if five login failure events (**RADIUS_LOGIN_FAIL**, **TELNET_LOGIN_FAIL**, or **SSH_LOGIN_FAIL**) are generated within 120 seconds from username **roger**. Take action by executing the login-fail.xsl event script, which disables the **roger** user account.

```
[edit event-options]
policy p2 {
   events [ RADIUS_LOGIN_FAIL TELNET_LOGIN_FAIL SSH_LOGIN_FAIL ];
   within 120 {
      trigger after 4;
   }
   attributes-match {
      RADIUS_LOGIN_FAIL.username matches roger;
      TELNET_LOGIN_FAIL.username matches roger;
   }
   then {
      event-script login-fail.xsl {
         destination some-dest;
      }
   }
}
```

Example: Ignoring Events Based on Receipt of Other Events

In the following policy, if any of ***event1***, ***event2***, or ***event3*** has occurred, and either ***event4*** or ***event5*** has occurred within the last 600 seconds, and ***event6*** has not occurred within the last 800 seconds, then the event that triggered the policy (***event1***, ***event2***, or ***event3***) is ignored, meaning system log messages are not created.

```
[edit event-options]
policy 1 {
   events [ event1 event2 event3 ];
   within 600 events [ event4 event5 ];
   within 800 not events event6;
   then {
      ignore;
   }
}
```

Sometimes events are generated repeatedly within a short period of time. In this case, it is redundant to execute a policy multiple times, once for each instance of the event. Event dampening allows you to slow down the execution of policies by ignoring instances of an event that occur within a specified time after another instance of the same event.

In the following example, an action is taken only if the eventd process has not received another instance of the event within the past 60 seconds. If an instance of the event has been received within the last 5 seconds, the policy is not executed and a system log message for the event is not created again.

```
[edit event-options]
policy dampen-policy {
   events event1;
   within 60 events event1;
```

```
    then {
      ignore;
    }
  }
  policy policy {
    events event1;
    then {
      ... actions ...
    }
  }
```

Example: Correlating Events Based on Event Attributes

In the following policy, the two events are correlated only if two of their parameter values match. Matching on attributes of both events ensures that the two events are related. In this case, the interface addresses must match and the physical interface (ifd) names must match.

The **RPD_KRT_IFDCHANGE** error occurs when the routing protocol process (rpd) sends a request to the kernel to change the state of an interface and the request fails. The **RPD_RDISC_NOMULTI** error occurs when an interface is configured for router discovery but the interface does not support IP multicast operations as required.

In this example, **RPD_RDISC_NOMULTI.interface-name** might be **so-0/0/0.0**, and **RPD_KRT_IFDCHANGE.ifd-index** might be **so-0/0/0**.

```
[edit event-options]
policy 1 {
  events rpd_rdisc_nomulti;
  within 500 events rpd_krt_ifdchange;
  attributes-match {
    rpd_rdisc_nomulti.interface-address equals rpd_krt_ifdchange.address;
    rpd_rdisc_nomulti.interface-name starts-with rpd_krt_ifdchange.ifd-index;
  }
  then {
    ... actions ...
  }
}
```

Controlling Event Policy Using a Regular Expression

The following policy is executed only if the **interface-name** attribute in both traps (**SNMP_TRAP_LINK_DOWN** and **SNMP_TRAP_LINK_UP**) match each other and the **interface-name** attribute in the **SNMP_TRAP_LINK_DOWN** trap starts with letter *t*. This means the policy is executed only for T1 (**t1-**) and T3 (**t3-**) interfaces. The policy is not executed when the **eventd** process receives traps from other interfaces.

NOTE: In system log files, the message tags appear in all uppercase letters. In the command-line interface (CLI), the message tags appear in all lowercase letters.

```
[edit event-options]
```

```
policy pol6 {
  events snmp_trap_link_down;
  within 120 events snmp_trap_link_up;
  attributes-match {
    snmp_trap_link_up.interface-name equals snmp_trap_link_down.interface-name;
    snmp_trap_link_down.interface-name matches "^t";
  }
  then {
    execute-commands {
      commands {
        "show interfaces {$$.interface-name}";
        "show configuration interfaces {$$.interface-name}";
      }
      output-filename config.txt;
      destination bsd2;
      output-format text;
    }
  }
}
```

Example: Generating an Internal Event Every Hour

In the following example, the internal event called **EVERY-ONE-HOUR** is generated every hour (3600 seconds). If 3601 seconds pass and the event has not been generated, certain actions are taken.

```
[edit event-options]
generate-event every-one-hour time-interval 3600;
policy check-heartbeat {
  events every-one-hour;
  within 3601 not events every-one-hour;
  then {
    ... actions ...
  }
}
```

Example: Generating an Internal Event at Midnight

In the following example, the internal event called **IT-IS-MIDNIGHT** is generated at 12:00 AM every night (00:00:00). When the eventd process receives the **IT-IS-MIDNIGHT** event, certain actions are taken.

```
[edit event-options]
generate-event it-is-midnight time-of-day 00:00:00;
policy midnight-chores {
  events it-is-midnight;
  then {
    ... actions ...
  }
}
```

Example: Raising an SNMP Trap in Response to an Event

Raise a trap and execute an associated event script in response to an event:

```
[edit event-options]
policy p1 {
   events ui_mgd_terminate;
   then {
      raise-trap;
      event-script bgp.xsl {
         arguments {
            destination {$ui_mgd_terminate.destination};
            code 2;
         }
         output-filename bgp-out;
         destination bsd3;
      }
   }
}
```

Example: Using Nonstandard System Log Messages to Trigger an Event Policy

Reference a **KERNEL** system log message in an event policy. The **raise-trap** action in the **then** statement is executed only if a **KERNEL** event containing a message that matches "exited on signal 11" occurs.

```
[edit event-options]
policy kernel-policy {
   events KERNEL;
   attributes-match {
      KERNEL.message matches "exited on signal 11";
   }
   then {
      raise-trap;
   }
}
```

CHAPTER 25

Summary of Event Policy Configuration Statements

This chapter describes each configuration statement for event policies. The statements are organized alphabetically.

archive-sites

Syntax

```
archive-sites {
  url <password password>;
}
```

Hierarchy Level [edit **event-options destinations** *destination-name*]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Specify an archive site to which files are transferred. If you specify more than one archive site, the device attempts to transfer to the first archive site in the list, moving to the next site only if the transfer fails.

Options ***url***—The archive destination specified as Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification. URLs of the type **file://** are not supported; however, local device directories are supported (for example, **/var/tmp/**).

password *password*—A plain-text password for login into the archive site.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Defining Destinations for File Archiving by Event Policies on page 377

arguments

Syntax

```
arguments {
  argument-name argument-value;
}
```

Hierarchy Level [edit **event-options policy** *policy-name* **then event-script** *filename*]

Release Information Statement introduced in Junos OS Release 7.6.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Define command-line arguments for an event script that is invoked from an event policy.

Options ***argument-name***—Name of the argument.

argument-value—Value of the argument.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Executing Event Scripts in an Event Policy on page 384

attributes-match

Syntax

```
attributes-match {
    event1.attribute-name equals event2.attribute-name;
    event.attribute-name matches regular-expression;
    event1.attribute-name starts-with event2.attribute-name;
}
```

Hierarchy Level [edit **event-options policy** *policy-name*]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Execute the policy only if the attributes of two events are correlated or if the attribute of one event matches a regular expression.

If the **attributes-match** statement includes the **equals** or **starts-with** options, or if it includes a **matches** option that includes a clause for an event that is not specified at the **[edit event-options policy *policy-name* events]** hierarchy level, you must include one or more **within** statements in the same policy configuration.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Using Correlated Events to Trigger an Event Policy on page 371
- **within on page 433**

commands

Syntax

```
commands {
  "command";
}
```

Hierarchy Level [edit **event-options policy** *policy-name* **then execute-commands**]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Specify an operational mode command to be issued on receipt of an event.

Options ***command***—Command to be issued. Enclose each command in quotation marks (“ ”). The event process (eventd) issues the commands in the order in which they appear in the configuration.

You can include variables in commands. The eventd process replaces each variable with values contained in the event that triggers the policy. You can use command variables of the following forms:

- **{$$.*attribute-name*}**—The double dollar sign (**$$**) notation represents the event that is triggering a policy. When combined with an attribute name, the command variable is replaced by the value of the attribute name of the triggering event.
- **{$*event*.*attribute-name*}**—The dollar sign with the event name (**$*event***) notation represents the most recent event that matches the specified event. The variable is replaced by the value of the attribute name of the most recent event that matches ***event***.
- **{$*.*attribute-name*}**—The dollar sign with the asterisk (**$***) notation represents the most recent event that matches any of the correlating events. The variable is replaced by the value of the attribute name of the most recent event that matches any of the events specified in the policy configuration.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring an Event Policy to Execute Operational Mode Commands on page 381
- Representing the Correlating Event in an Event Policy on page 373

destination

Syntax

```
destination destination-name {
    retry-count count retry-interval seconds;
    transfer-delay seconds;
}
```

Hierarchy Level

[edit **event-options policy** *policy-name* **then event-script** *filename*],
[edit **event-options policy** *policy-name* **then execute-commands**]

Release Information

Statement introduced in Junos OS Release 7.5.
Support extended to the **[edit event-options policy** ***policy-name*** **then event-script** ***filename*****]** hierarchy level in Junos OS Release 7.6.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description

Assign a location to which to upload command or script output for the specified policy.

Options

destination-name—Name of a destination defined in the **destinations** statement at the **[edit event-options]** hierarchy level.

The remaining statements are explained separately.

Required Privilege Level

maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring an Event Policy to Execute Operational Mode Commands on page 381
- Executing Event Scripts in an Event Policy on page 384
- **destinations on page 412**

destinations

Syntax

```
destinations {
  destination-name {
    archive-sites {
      url <password password>;
    }
    transfer-delay seconds;
  }
}
```

Hierarchy Level [edit event-options]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Define one or more destinations, each with a unique name and other attributes. You can use the destination as a storage location for command output and for various files, such as system log files and core files.

Options ***destination-name***—Name of a destination.

The remaining statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Defining Destinations for File Archiving by Event Policies on page 377

equals

Syntax *event1.attribute-name* equals *event2.attribute-name*;

Hierarchy Level [edit **event-options policy** *policy-name* **attributes-match**]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Execute the policy only if the specified attribute of ***event1*** equals the specified attribute of ***event2***.

Options ***event1.attribute-name***—Attribute of one event.

event2.attribute-name—Attribute of another event.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Using Correlated Events to Trigger an Event Policy on page 371

event-options

Syntax

```
event-options {
    destinations {
        destination-name {
            archive-sites {
                url <password password>;
            }
            transfer-delay seconds;
        }
    }
    event-script {
        file filename {
            checksum (md5 | sha-256 | sha1) hash;
            refresh;
            refresh-from url;
            remote-execution {
                remote-hostname {
                    passphrase user-password;
                    username user-login;
                }
            }
            source url;
        }
        refresh;
        refresh-from url;
        traceoptions {
            file <filename> <files number> <size size> <world-readable | no-world-readable>;
            flag flag;
            no-remote-trace;
        }
    }
    generate-event event-name {
        time-interval seconds;
        time-of-day hh:mm:ss;
    }
    policy policy-name {
        attributes-match {
            event1.attribute-name equals event2.attribute-name;
            event.attribute-name matches regular-expression;
            event1.attribute-name starts-with event2.attribute-name;
        }
        events [ events ];
        within seconds not events [ events ];
        then {
            event-script filename {
                arguments {
                    argument-name argument-value;
                }
                output-filename filename;
                destination destination-name {
                    retry-count count retry-interval seconds;
                    transfer-delay seconds;
                }
```

```
        }
        execute-commands {
          commands {
            "command";
          }
          destination destination-name {
            retry-count count retry-interval seconds;
            transfer-delay seconds;
          }
          output-filename filename;
          output-format (text | xml);
          user-name username;
        }
        ignore;
        raise-trap;
        upload filename (filename | committed) destination destination-name {
          retry-count count retry-interval seconds;
          transfer-delay seconds;
          user-name username;
        }
      }
    }
    traceoptions {
      file filename <files number> <size size> <world-readable | no-world-readable>;
      flag flag;
    }
}
```

Hierarchy Level [edit]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Configure event policies.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

event-script

Syntax

```
event-script filename {
  arguments {
    argument-name argument-value;
  }
  destination destination-name {
    retry-count count retry-interval seconds;
    transfer-delay seconds;
  }
  output-filename filename;
  output-format (text | xml);
  user-name username;
}
```

Hierarchy Level [edit **event-options policy** *policy-name* **then**]

Release Information Statement introduced in Junos OS Release 7.6.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description On receipt of an event, specify operational mode commands to be issued, the format of the command output, and a name and destination for the output file.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Executing Event Scripts in an Event Policy on page 384

events

See the following sections:

- **events (Associating Events with a Policy) on page 417**
- **events (Correlating Events with Each Other) on page 417**

events (Associating Events with a Policy)

Syntax events [*events*];

Hierarchy Level [edit **event-options policy** *policy-name*]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Create a list of events that trigger this policy. If one or more of the listed events occurs, the policy is executed.

Options **[*events*]**—List of events. Events can be internally generated, or they can be generated by Junos OS processes.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Using Correlated Events to Trigger an Event Policy on page 371

events (Correlating Events with Each Other)

Syntax events [*events*];

Hierarchy Level [edit **event-options policy** *policy-name* **within** *seconds*]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Create a list of events that must occur within a specified time interval for the policy to be triggered.

Options **[*events*]**—List of events. Events can be internally generated, or they can be generated by Junos OS processes.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Using Correlated Events to Trigger an Event Policy on page 371

execute-commands

Syntax

```
execute-commands {
  commands {
    "command";
  }
  destination destination-name {
    retry-count count retry-interval seconds;
    transfer-delay seconds;
  }
  output-filename filename;
  output-format (text | xml);
  user-name username;
}
```

Hierarchy Level [edit **event-options policy** *policy-name* **then**]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description On receipt of an event, specify operational mode commands to be issued, the format of the command output, and a name and destination for the output file.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring an Event Policy to Execute Operational Mode Commands on page 381

generate-event

Syntax

```
generate-event event-name {
    time-interval seconds;
    time-of-day hh:mm:ss;
}
```

Hierarchy Level [edit event-options]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Generate an internal event, based on a time interval or the time of day.

Options ***event-name***—Name of an internally generated event.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Generating Internal Events to Trigger Event Policies on page 375

ignore

Syntax

```
ignore;
```

Hierarchy Level [edit event-options policy *policy-name* then]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Define a policy that ignores particular events. If one or more of the listed events occur, a system log message for the event is not generated, and no further policies associated with this event are processed. If you include the **ignore** statement in a policy configuration, you cannot configure any other actions in the policy.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring Event Policies to Ignore an Event on page 389

matches

Syntax *event.attribute-name* matches *regular-expression*;

Hierarchy Level [edit **event-options policy** *policy-name* **attributes-match**]

Release Information Statement introduced in Junos OS Release 7.5.

Description Execute the policy only if the specified attribute of ***event*** matches a regular expression.

Options ***event.attribute-name***—Event attribute to compare to a regular expression.

regular-expression—Regular expression to compare.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Using Correlated Events to Trigger an Event Policy on page 371
- Using Regular Expressions to Refine the Set of Events That Trigger a Policy on page 374

not

Syntax not events [*events*];

Hierarchy Level [edit **event-options policy** *policy-name* **within** *seconds*]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Create a list of events that must not occur within the specified time interval for the policy to be triggered.

Options **[*events*]**—List of events. Events can be internally generated, or they can be generated by Junos OS processes.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Using Correlated Events to Trigger an Event Policy on page 371

output-filename

Syntax output-filename *filename*;

Hierarchy Level [edit **event-options policy** *policy-name* **then event-script** *filename*],
[edit **event-options policy** *policy-name* **then execute-commands**]

Release Information Statement introduced in Junos OS Release 7.5.
Support at the **[edit event-options policy *policy-name* then event-script *filename*]** hierarchy level introduced in Junos OS Release 7.6.

Description Assign a filename to which to write command or script output for the specified commands or script. For op scripts, this statement is optional.

Options ***filename***—Name of a file in which to write command or script output.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Configuring an Event Policy to Execute Operational Mode Commands on page 381
- Executing Event Scripts in an Event Policy on page 384

output-format

Syntax output-format (text | xml);

Hierarchy Level [edit **event-options policy** *policy-name* **then event-script** *filename*],
[edit **event-options policy** *policy-name* **then execute-commands**]

Release Information Statement introduced in Junos OS Release 7.5.
Support at the **[edit event-options policy** ***policy-name*** **then event-script** ***filename*****]** hierarchy level introduced in Junos OS Release 8.3.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Specify the format (ASCII text or XML) for the output of the specified commands or script.

Options **text**—Formatted ASCII text.

xml—Junos Extensible Markup Language (XML) tags.

Default: **xml** at the **[edit event-options policy** ***policy-name*** **then execute-commands]** hierarchy level and **text** at the **[edit event-options policy** ***policy-name*** **then event-script** ***filename*****]** hierarchy level.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring an Event Policy to Execute Operational Mode Commands on page 381
- Executing Event Scripts in an Event Policy on page 384

policy

Syntax

```
policy policy-name {
  attributes-match {
    event1.attribute-name equals event2.attribute-name;
    event.attribute-name matches regular-expression;
    event1.attribute-name starts-with event2.attribute-name;
  }
  events [ events ];
  then {
    ... the then subhierarchy appears at the end of the [edit event-options policy policy-name]
      hierarchy level ...
  }
  within seconds {
    events [ events ];
    not events [ events ];
    trigger (on | after | until) event-count;
  }

  then {
    event-script filename {
      arguments {
        argument-name argument-value;
      }
      destination destination-name {
        retry-count count retry-interval seconds;
        transfer-delay seconds;
      }
      output-filename filename;
      output-format (text | xml);
      user-name username;
    }
    execute-commands {
      commands {
        "command";
      }
      destination destination-name {
        retry-count count retry-interval seconds;
        transfer-delay seconds;
      }
      output-filename filename;
      output-format (text | xml);
      user-name username;
    }
    ignore;
    raise-trap;
    upload filename (filename | committed) destination destination-name {
      retry-count count retry-interval seconds;
      transfer-delay seconds;
      user-name username;
    }
  }
}
```

Hierarchy Level [edit **event-options**]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Define an event policy to be processed by the eventd process. If you configure a policy, the **events** and **then** statements are mandatory.

You can configure multiple policies to be processed for an event. The policies are executed in the order in which they appear in the configuration. If you configure more than one policy for an event, and if one of the policies is to ignore the event, no policies that follow the **ignore** statement are executed.

Default If you do not configure a policy for an event, the event is recorded in the system log.

Options ***policy-name***—Name of an event policy.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

raise-trap

Syntax raise-trap;

Hierarchy Level [edit **event-options policy** *policy-name* **then**]

Release Information Statement introduced in Junos OS Release 8.1.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Define a policy that raises an SNMP trap in response to an event. If one or more of the listed events occur, the system log message for the event is converted into a trap. This enables an agent to notify a trap-based network management system (NMS) of significant events.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring Event Policies to Raise SNMP Traps on page 390

retry-count

Syntax retry-count *number* retry-interval *seconds*;

Hierarchy Level [edit **event-options policy** *policy-name* **then event-script** *filename* **destination** *destination-name*],
[edit **event-options policy** *policy-name* **then execute-commands destination** *destination-name*],
[edit **event-options policy** *policy-name* **then upload** filename (*filename* | committed) destination *destination-name*]

Release Information Statement introduced in Junos OS Release 8.4.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Configure an event policy to retry a file upload operation if the first attempt fails.

Default If you do not include this statement, the file upload operation is attempted one time only.

Options ***number***—Number of retries.

retry-interval *seconds*—Length of time to wait between retries.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Configuring an Event Policy to Retry the File Upload Action on page 380

starts-with

Syntax *event1.attribute-name* starts-with *event2.attribute-name*;

Hierarchy Level [edit **event-options policy** *policy-name* **attributes-match** *event1.attribute-name*]

Release Information Statement introduced in Junos OS Release 7.5.

Description Execute the policy only if the specified attribute of ***event1*** starts with the specified attribute of ***event2***.

Options ***event1.attribute-name***—Attribute of one event.

event2.attribute-name—Attribute of another event.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Using Correlated Events to Trigger an Event Policy on page 371

then

Syntax

```
then {
  event-script filename {
    arguments {
      argument-name argument-value;
    }
    destination destination-name {
      retry-count count retry-interval seconds;
      transfer-delay seconds;
    }
    output-filename filename;
    output-format (text | xml);
    user-name username;
  }
  execute-commands {
    commands {
      "command";
    }
    destination destination-name {
      retry-count count retry-interval seconds;
      transfer-delay seconds;
    }
    output-filename filename;
    output-format (text | xml);
    user-name username;
  }
  ignore;
  raise-trap;
  upload filename (filename | committed) destination destination-name {
    retry-count count retry-interval seconds;
    transfer-delay seconds;
    user-name username;
  }
}
```

Hierarchy Level [edit event-options policy *policy-name*]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Define actions to take if an event occurs. For each policy, you can configure multiple actions.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring an Event Policy to Upload Files on page 377
- Configuring an Event Policy to Execute Operational Mode Commands on page 381
- Executing Event Scripts in an Event Policy on page 384

- Configuring Event Policies to Ignore an Event on page 389
- Configuring Event Policies to Raise SNMP Traps on page 390

time-interval

Syntax time-interval *seconds*;

Hierarchy Level [edit **event-options generate-event** *event-name*]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Configure a frequency at which to generate a particular event.

Options ***seconds***—Time interval between internally generated events.
Range: 60 through 2,592,000 seconds

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Generating Internal Events to Trigger Event Policies on page 375

time-of-day

Syntax time-of-day *hh*:*mm*:*ss*;

Hierarchy Level [edit **event-options generate-event** *event-name*]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Configure a time of day at which to generate a particular event.

Options ***hh*:*mm*:*ss***—Time of day at which to generate an event.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Generating Internal Events to Trigger Event Policies on page 375

traceoptions

Syntax

```
traceoptions {
  file <filename> <files number> <match regular-expression> <size size> <world-readable |
    no-world-readable>;
  flag flag;
  no-remote-trace;
}
```

Hierarchy Level [edit **event-options**]

Release Information Statement introduced in Junos OS Release 7.5.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Define tracing operations for event policies.

Default If you do not include this statement, no event-policy-specific tracing operations are performed.

Options ***filename***—Name of the file to receive the output of the tracing operation. All files are placed in the directory **/var/log**. By default, commit script process tracing output is placed in the file **eventd**. If you include the **file** statement, you must specify a filename. To retain the default, you can specify **eventd** as the filename.

files ***number***—(Optional) Maximum number of trace files. When a trace file named *trace-file* reaches its maximum size, it is renamed and compressed to *trace-file*.0.gz. When *trace-file* again reaches its maximum size, *trace-file*.0.gz is renamed *trace-file*.1.gz and *trace-file* is renamed and compressed to *trace-file*.0.gz. This renaming scheme continues until the maximum number of trace files is reached. Then the oldest trace file is overwritten.

If you specify a maximum number of files, you also must specify a maximum file size with the **size** option and a filename.

Range: 2 through 1000

Default: 3 files

flag—Tracing operation to perform. To specify more than one tracing operation, include multiple **flag** statements. You can include the following flags:

- **all**—Log all operations
- **configuration**—Log reading of configuration at the **[edit event-options]** hierarchy level
- **events**—Log eventd processing
- **database**—Log events involving storage and retrieval in events database
- **server**—Log communication with processes that are generating events
- **timer-events**—Log internally generated events

match *regular-expression*—(Optional) Refine the output to include lines that contain the regular expression.

no-world-readable—Restrict file access to owner. This is the default.

size *size*—(Optional) Maximum size of each trace file, in kilobytes (KB), megabytes (MB), or gigabytes (GB). When a trace file named *trace-file* reaches this size, it is renamed and compressed to *trace-file*.0.gz. When the *trace-file* again reaches its maximum size, *trace-file*.0.gz is renamed *trace-file*.1.gz and *trace-file* is renamed and compressed to *trace-file*.0.gz. This renaming scheme continues until the maximum number of trace files is reached. Then the oldest trace file is overwritten.

If you specify a maximum file size, you also must specify a maximum number of trace files with the **files** option and filename.

Syntax: ***x*k** to specify KB, ***x*m** to specify MB, or ***x*g** to specify GB

Range: 10 KB through 1 GB

Default: 128 KB

world-readable—(Optional) Enable unrestricted file access.

Required Privilege Level

maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Tracing Event Policy Processing on page 391

transfer-delay

Syntax transfer-delay *seconds*;

Hierarchy Level [edit **event-options destinations** *destination-name*],
[edit **event-options policy** *policy-name* **then event-script** *filename* **destination** *destination-name*],
[edit **event-options policy** *policy-name* **then execute-commands destination** *destination-name*],
[edit **event-options policy** *policy-name* **then upload** filename (*filename* | committed) destination *destination-name*]

Release Information Statement introduced in Junos OS Release 7.5.
Support at the **[edit event-options policy *policy-name* then ...]** hierarchy levels introduced in Junos OS Release 8.4.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Configure a delay before transferring files. This allows the files to be completely generated before the upload starts. If you configure a transfer delay at the **[edit event-options destination *destination-name*]** hierarchy level and at one of the **[edit event-options policy *policy-name* then ...]** hierarchy levels, the resulting delay is the sum of the two delays.

Default If you do not include this statement, there is no transfer delay.

Options ***seconds***—Duration of the delay before files are uploaded.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Defining Destinations for File Archiving by Event Policies on page 377
- Configuring the Delay Before Files Are Uploaded by an Event Policy on page 379

trigger

Syntax trigger (on | after | until) *event-count*;

Hierarchy Level [edit **event-options policy** *policy-name* **within** *seconds*]

Release Information Statement introduced in Junos OS Release 8.4.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Configure an event policy to be triggered if an event or set of events occurs ***event-count*** times within a specified time period.

Default If you do not include this statement, the policy is executed on receipt of the first configured event.

Options

after *event-count*—The policy is executed when the number of matching events received equals ***event-count*** + 1.

on *event-count*—The policy is executed when the number of matching events received equals ***event-count***.

until *event-count*—The policy is executed each time a matching event is received and stops being executed when the number of matching events received equals ***event-count***.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Triggering an Event Policy Based on Event Count on page 374

upload

Syntax

```
upload filename  (filename | committed) destination destination-name {
    retry-count count retry-interval seconds;
    transfer-delay seconds;
    user-name username;
}
```

Hierarchy Level [edit **event-options policy** *policy-name* **then**]

Release Information Statement introduced in Junos OS Release 7.5.
committed option to **filename** statement introduced in Junos OS Release 8.1.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description On receipt of an event, upload the committed configuration file to a destination.

Options

destination *destination-name*—Name of the destination for the uploaded file. It must be defined in the **destinations** statement at the **[edit event-options]** hierarchy level.

filename (*filename* | committed)—Name of the file to upload. Specify either the word **committed** to upload the most recently committed configuration file, or the filename of another file.

The remaining statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- **destinations on page 412**
- Configuring an Event Policy to Upload Files on page 377

user-name

Syntax user-name *username*;

Hierarchy Level [edit **event-options policy** *policy-name* **then event-script** *filename*],
[edit **event-options policy** *policy-name* **then execute-commands**],
[edit **event-options policy** *policy-name* **then upload** filename (*filename* | committed) destination *destination-name*]

Release Information Statement introduced in Junos OS Release 8.4.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Associate a user with an action in an event policy. The event policy action is executed under the privileges of the associated user.

Default If you do not associate a user with an action, the action is executed as user **root**.

Options ***username***—A username that is configured at the **[edit system login]** hierarchy level.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Changing the User Privilege Level for an Event Policy Action on page 390

within

Syntax
```
within seconds {
  events [ events ];
  not events [ events ];
  trigger (after | on | until) event-count;
}
```

Hierarchy Level [edit **event-options policy** *policy-name*]

Release Information Statement introduced in Junos OS Release 7.5.

Description Create a list of events that must (or must not) occur within a specified time interval for the policy to be triggered.

The statements are explained separately.

Options ***seconds***—Interval between events.

Range: 60 through 604,800 seconds

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Using Correlated Events to Trigger an Event Policy on page 371

PART 5

Event Automation

CHAPTER 26

Event Scripts Overview

This chapter discusses the following topics:

Event Script Programming Overview

Junos OS event scripts are triggered automatically by defined event policies in response to a system event and can instruct the Junos OS to take immediate action. Event scripts automate network and device management and troubleshooting. Event scripts can perform functions available through the remote procedure calls (RPCs) supported by either Junos XML management protocol or the Junos Extensible Markup Language (XML) API. Event scripts are executed by the event process (eventd).

Event scripts allow you to do the following things:

- Diagnose and fix network problems automatically.
- Monitor the overall status of a device.
- Reconfigure the device to avoid or work around known problems in the Junos OS.
- Change the device's configuration in response to a problem.

Event scripts are based on the Junos XML management protocol and the Junos XML API, which are discussed in "Junos XML API and Junos XML Management Protocol Overview" on page 15. Event scripts can be written in either the Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) scripting language. Event scripts use XPath to locate the operational objects to be inspected and XSLT constructs to specify the actions to perform on the located operational objects. The actions can change the output or execute additional commands based on the output. For more information about XPath and XSLT, see "XPath Overview" on page 21 and "XSLT Overview" on page 19. For more information about SLAX, see "SLAX Overview" on page 35.

How Event Scripts Work

Event scripts initiate operational commands when triggered by an event policy. When an event policy is triggered, this policy forwards event details to the event script. You

enable event scripts by listing the names of one or more event script files within the **[edit event-options event-script]** hierarchy level. These scripts contain instructions that execute operational mode commands and inspect the output automatically. Event scripts are invoked within an event policy. For information about event policies, see "Event Notifications and Policies Overview" on page 365 and "Executing Event Scripts in an Event Policy" on page 384.

You can use event scripts to generate changes to the device configuration by including the **<load-configuration>** tag element. Because the changes are loaded before the standard validation checks are performed, they are validated for correct syntax, just like statements already present in the configuration before the script is applied. If the syntax is correct, the configuration is activated and becomes the active, operational device configuration.

CHAPTER 27

Writing Event Scripts

This chapter explains how to write event scripts and includes the following topics:

Required Boilerplate for Event Scripts

When you write event scripts, you use Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) tools provided with the Junos OS. These tools include basic boilerplate that you must include in all event scripts, optional extension functions that accomplish scripting tasks more easily, and named templates that make scripts easier to read and write, which you import from a file called junos.xsl. For more information about the extension functions and templates, see "Junos Extension Functions in the jcs Namespace" on page 62.

Event scripts are based on Junos XML and Junos XML protocol tag elements. Like all XML elements, angle brackets enclose the name of a Junos XML or Junos XML protocol tag element in its opening and closing tags. This is an XML convention, and the brackets are a required part of the complete tag element name. They are not to be confused with the angle brackets used in the documentation to indicate optional parts of Junos OS CLI command strings.

You must include either XSLT or SLAX boilerplate as the starting point for all event scripts that you create. The XSLT boilerplate follows:

XSLT Boilerplate for Event Scripts

```
1  <?xml version="1.0" standalone="yes"?>
2  <xsl:stylesheet version="1.0"
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4     xmlns:junos="http://xml.juniper.net/junos/*/junos"
5     xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
6     xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
7     <xsl:import href="../import/junos.xsl"/>

8     <xsl:template match="configuration">
```

```
 9        <event-script-results>
           <!-- ... Insert your code here ... -->
10        </event-script-results>
11     </xsl:template>
       <!-- ... insert additional template definitions here ... -->
12  </xsl:stylesheet>
```

Line 1 is the Extensible Markup Language (XML) processing instruction (PI). This PI specifies that the code is written in XML using version 1.0. The XML PI, if present, must be the first noncomment token in the script file.

```
1  <?xml version="1.0"?>
```

Line 2 opens the style sheet and specifies the XSLT version as 1.0.

```
2  <xsl:stylesheet version="1.0"
```

Lines 3 through 6 list all the namespace mappings commonly used in event scripts. Not all of these prefixes are used in this example, but it is not an error to list namespace mappings that are not referenced. Listing all namespace mappings prevents errors if the mappings are used in later versions of the script.

```
3     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4     xmlns:junos="http://xml.juniper.net/junos/*/junos"
5     xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
6     xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
```

Line 7 is an XSLT import statement. It loads the templates and variables from the file referenced as **../import/junos.xsl**, which ships as part of the Junos OS (in the file /usr/libdata/cscript/import/junos.xsl). The junos.xsl file contains a set of named templates you can call in your scripts. These named templates are discussed in "Junos Named Templates in the jcs Namespace" on page 82.

```
7     <xsl:import href="../import/junos.xsl"/>
```

Line 8 defines a template that matches the **</>** element. The **<xsl:template match="/">** element is the root element and represents the top level of the XML hierarchy. All XML Path Language (XPath) expressions in the script must start at the top level. This allows the script to access all possible Junos XML and Junos XML protocol remote procedure calls (RPCs). For more information, see "XPath Overview" on page 21 and **xsl:template match="/" Template**.

```
8     <xsl:template match="/">
```

After the **<xsl:template match="/">** tag element, the **<event-script-results>** and **</event-script-results>** container tags must be the top-level child tags, as shown in Lines 9 and 10.

```
9        <event-script-results>
          <!-- ... insert your code here ... -->
10       </event-script-results>
```

Line 11 closes the template.

```
11    </xsl:template>
```

Between Line 11 and Line 12, you can define additional XSLT templates that are called from within the **<xsl:template match="/">** template.

Line 12 closes the style sheet and the event script.

```
12  </xsl:stylesheet>
```

SLAX Boilerplate for Event Scripts

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";

match / {
  <event-script-results> {
    /*
    * Insert your code here
    */
  }
}
```

Mapping Operational Mode Commands and Output Fields to Junos XML Notation

In event scripts, you use tag elements from the Junos XML API to represent operational mode commands and output fields. For the Junos XML equivalent of commands and output fields, consult the *Junos XML API Operational Reference*.

You can also display Junos XML by directing the output from the **show** command to the **| display xml** command:

```
user@host> operational-mode-command | display xml
```

For example:

```
user@host> show interfaces terse | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/junos">
    <interface-information
xmlns="http://xml.juniper.net/junos/10.0RI0/junos-interface" junos:style="terse">
        <physical-interface>
            <name>dsc</name>
            <admin-status>up</admin-status>
            <oper-status>up</oper-status>
        </physical-interface>
        <physical-interface>
            <name>fxp0</name>
            <admin-status>up</admin-status>
            <oper-status>up</oper-status>
            <logical-interface>
                <name>fxp0.0</name>
                <admin-status>up</admin-status>
                <oper-status>up</oper-status>
                ...
```

Using RPCs and Operational Mode Commands in Event Scripts

Most Junos operational mode commands have XML equivalents. These XML commands can be executed remotely using the *remote procedure call* (RPC) protocol. All operational mode commands that have XML equivalents are listed in the *Junos XML API Operational Reference*.

RPC and operational mode command use in event scripts is discussed in more detail in the following sections:

Using RPCs in Event Scripts

You can invoke remote procedure calls (RPCs) in event scripts. For each event script that invokes RPCs, you must include the **remote-execution** statement at the **[edit event-options event-script file *filename*]** hierarchy level. For each remote device where an RPC is executed, you must configure the SSH host key information for the that device on the local device where the event script is executed.

For each remote device where an RPC is executed, specify the device hostname and the corresponding username and passphrase at the **remote-execution** level of the configuration hierarchy.

```
[edit event-options event-script file filename]
remote-execution {
  remote-hostname {
    username username;
    passphrase passphrase;
  }
}
```

The remote hostnames and their corresponding username and passphrase, in addition to the event details, are passed as input to the event script when it is triggered by an event policy. For more information about the details that are forwarded to the event script, see “Capturing and Using Event Details and Remote Execution Details in Event Scripts” on page 445. A connection handle to the remote host is generated with the **jcs:open()** function using ***remote-hostname***, ***username***, and ***passphrase*** as arguments; for more information about this function, see **jcs:open() Function**. The following code obtains a connection handle for each remote host included in the configuration:

XSLT Syntax

```
<xsl:for-each select="event-script-input/remote-execution-details">
  <xsl:variable name="d" select="remote-execution-detail"/>
  <xsl:variable name="connection"
      select="jcs:open($d/remote-hostname,$d/username,$d/passphrase)"/>
  ...
</xsl:for-each>
```

SLAX Syntax

```
for-each (event-script-input/remote-execution-details) {
  var $d = remote-execution-detail;
  var $connection = jcs:open($d/remote-hostname,$d/username,$d/passphrase);
  ...
}
```

To execute an RPC on a remote device, an SSH session must be established. In order for the script to establish the connection, you must either configure the SSH host key information for the remote device on the local device where the script will be executed, or the SSH host key information for the remote device must exist in the known hosts file

of the user executing the script. For each remote device where the RPC is executed, configure the SSH host key information with one of the following methods:

- To configure SSH known hosts on the local device, include the **host** statement, and specify hostname and host key options for the remote device at the **[edit security ssh-known-hosts]** hierarchy level of the configuration.
- To manually retrieve SSH host key information, issue the **set security ssh-known-hosts fetch-from-server *hostname*** configuration mode command to instruct the Junos OS to connect to the remote device and add the key.

```
user@host# set security ssh-known-hosts fetch-from-server router2

The authenticity of host 'router2 (10.10.10.1)' can't be established.
RSA key fingerprint is 30:18:99:7a:3c:ed:40:04:0f:fd:c1:57:7e:6b:f3:90.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'router2,10.10.10.1' (RSA) to the list of known
 hosts.
```

- To manually import SSH host key information from a file, use the **set security ssh-known-hosts load-key-file *filename*** configuration mode command and specify the known-hosts file.

```
user@host# set security ssh-known-hosts load-key-file /var/tmp/known_hosts

Import SSH host keys from trusted source /var/tmp/known_hosts ? [yes,no]
 (no) yes
```

- Alternatively, the user executing the script can log in to the local device, SSH to the remote device, and then manually accept the host key, which is added to that user's known hosts file. In the following example, root is logged in to **router1**. In order to execute a remote RPC on **router2**, root adds the host key of **router2** by issuing the **ssh router2** operational mode command and manually accepting the key.

```
root@router1> ssh router2

The authenticity of host 'router2 (10.10.10.1)' can't be established.
RSA key fingerprint is 30:18:99:7a:3c:ed:40:04:0f:fd:c1:57:7e:6b:f3:90.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'router2,10.10.10.1' (RSA) to the list of known
 hosts.
```

After configuring the required SSH host key and obtaining a connection handle to the remote device, the event script can execute RPCs with the **jcs:execute()** extension function on that remote device. This function is described in **jcs:execute() Function**. To use an RPC in the event script, include the RPC in a variable declaration and execute it with the **jcs:execute()** function; the connection handle and RPC variable declaration are provided as arguments to the **jcs:execute()** function.

XSLT Syntax

```
<xsl:variable name="rpc">
  <get-interface-information/> # Junos RPC for the show interfaces command
</xsl:variable>
<xsl:variable name="out" select="jcs:execute($connection, $rpc)"/>
```

SLAX Syntax

```
var $rpc = <get-interface-information>;
var $out = jcs:execute($connection, $rpc);
```

where **$connection** is the connection handle to the remote host. Any number of RPCs can be executed within the context of this connection handle until it is closed with the **jcs:close()** function.

Displaying the RPC Tags for a Command

To display the remote procedure call (RPC) XML tags for an operational mode command, enter **display xml rpc** after the pipe symbol (|).

The following example displays the RPC tags for the **show route** command:

```
user@host> show route | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.1I0/junos">
    <rpc>
        <get-route-information>
        </get-route-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Using Operational Mode Commands in Event Scripts

Some operational mode commands do not have XML equivalents. If a command is not listed in the *Junos XML API Operational Reference*, it does not have an XML equivalent.

Another way to determine whether a command has an XML equivalent is to issue the command followed by the **| display xml** command:

```
user@host> operational-mode-command | display xml
```

If the output includes only tag elements like **<output>**, **<cli>**, and **<banner>**, the command might not have an XML equivalent. In the following example, the output indicates that the **show host** command has no XML equivalent:

```
user@host> show host hostname | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/junos">
   <output>
       ...
    </output>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

NOTE: For some commands that have an XML equivalent, the output of the piped **| display xml** command does not include tag elements other than **<output>**, **<cli>**, and **<banner>** only because the relevant feature is not configured. For example, the **show services cos statistics forwarding-class** command has an XML equivalent that returns output in the **<service-cos-forwarding-class-statistics>** response tag, but if the configuration does not include include any statements at the **[edit class-of-service]** hierarchy level then there is no actual data for the **show services cos statistics forwarding-class | display xml** command to display. The output is something like this:

```
user@host> show services cos statistics forwarding-class | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/8.3I0/junos">
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

For this reason, the information in the *Junos XML API Operational Reference* is normally more reliable.

An event script can include commands that have no XML equivalent. Use the **<command>**, **<xsl:value-of>**, and **<output>** elements in the script, as shown in the following code snippet. This snippet is expanded and fully described in “Example: Displaying DNS Hostname Information Using an Op Script” on page 337.

```
<xsl:variable name="query">
  <command>
    <xsl:value-of select="concat('show host ', $hostname)"/>
  </command>
</xsl:variable>
<xsl:variable name="result" select="jcs:invoke($query)"/>
<xsl:variable name="host" select="$result"/>
<output>
  <xsl:value-of select="concat('Name: ', $host)"/>
</output>
...
```

Capturing and Using Event Details and Remote Execution Details in Event Scripts

When an event script is triggered by an event policy, the initiating event policy forwards a set of event details to the triggered event script. These event details can be captured, evaluated, and sent to log files as required. In addition, any configured remote execution details are also forwarded to the event script. The remote execution details allow the event script to invoke remote procedure calls as detailed in “Using RPCs and Operational Mode Commands in Event Scripts” on page 441.

Two types of event details are returned: triggered events and received events. *Triggered events* record the details of the event that triggered the policy. *Received events* record the details of events that happened before the triggering event. Event details and remote execution details are forwarded to the event script as XML in the following format:

```
<event-script-input>
  <trigger-event>
    <id>event-id</id>
    <type>event-type</type>
    <generation-time>timestamp</generation-time>
    <process>
      <name>process-name</name>
      <pid>pid</pid>
    </process>
    <hostname>hostname</hostname>
    <facility>facility-string</facility>
    <severity>severity-string</severity>
    <attribute-list>
      <attribute>
        <name>attribute-name</name>
        <value>attribute-value</value>
      </attribute>
    </attribute-list>
  </trigger-event>
  <received-events>
    <received-event>
      <id>event-id</id>
      <type>event-type</type>
      <generation-time>timestamp</generation-time>
      <process>
        <name>process-name</name>
        <pid>pid</pid>
      </process>
      <hostname>hostname</hostname>
      <facility>facility-string</facility>
      <severity>severity-string</severity>
      <attribute-list>
        <attribute>
          <name>attribute-name</name>
          <<value>attribute-value</value>
        </attribute>
      </attribute-list>
    </received-event>
  </received-events>
  <remote-execution-details>
    <remote-execution-detail>
      <remote-hostname>hostname</remote-hostname>
      <username>username</username>
      <passphrase>passphrase</passphrase>
    </remote-execution-detail>
  </remote-execution-details>
</event-script-input>
```

For information about one method for using event details, see “Example: Limiting Event Script Output Based on a Specific Event Type” on page 459.

CHAPTER 28

Configuring Event Scripts

Event scripts allow you to automate network troubleshooting and network management. This chapter discusses command-line interface (CLI) configuration statements and operational mode commands for enabling and executing scripts. Much of this discussion is applicable mainly to op scripts; however, most of the CLI statements discussed have a Junos XML protocol counterpart, and thus the concepts discussed in this section are helpful for event scripts.

To configure event scripts, include the following statements at the **[edit event-options]** hierarchy level:

```
[edit event-options]
event-script {
    file filename {
        checksum (md5 | sha-256 | sha1) hash;
        refresh;
        refresh-from url;
        remote-execution {
            remote-hostname {
                passphrase user-password;
                username user-login;
            }
        }
        source url;
    }
    refresh;
    refresh-from url;
    traceoptions {
        file <filename> <files number> <size size> <world-readable | no-world-readable>;
        flag flag;
        no-remote-trace;
    }
}
```

This chapter discusses the following topics:

Implementing Event Scripts

This section provides directions for using event scripts on a device running Junos OS.

Installing Event Scripts on a Device Running Junos OS

To install event scripts on a device running Junos OS, follow these steps:

1. Write the event script. For information about writing event scripts, see "Writing Event Scripts" on page 439. For event script examples, see "Event Script Examples" on page 459.
2. Copy the script to the /var/db/scripts/event directory on the hard drive or the /config/scripts/event directory on the flash drive. For information about setting the storage location for scripts, see "Storing Event Scripts in Flash Memory" on page 451. Only users who belong to the Junos OS **super-user** login class can access and edit files in these directories.

   NOTE: If the device has dual Routing Engines and you want to enable the event script to execute on both Routing Engines, you must copy the script to the /var/db/scripts/event or /config/scripts/event directory on both Routing Engines. The commit synchronize command does not automatically copy scripts between Routing Engines.

3. Enable the script by including the **file *filename*** statement at the **[edit event-options event-script]** hierarchy level. For instructions, see "Enabling an Event Script" on page 449.
4. Issue the **commit** command.

After the commit operation is completed, the event script is loaded into memory and ready for automatic execution in response to system log events. For more information, see "Executing Event Scripts in an Event Policy" on page 384.

Replacing an Event Script

You can update or replace an existing event script without changing the device's configuration or disrupting operations. Follow these steps:

1. Edit or write the new event script.

   For information about writing event scripts, see "Writing Event Scripts" on page 439.

2. Copy the script to the /var/db/scripts/event directory on the hard drive or the /config/scripts/event directory on the flash drive; for information about setting the storage location for scripts, see “Storing Event Scripts in Flash Memory” on page 451. Only users who belong to the Junos **super-user** login class can alter files in these directories.

   NOTE: If the device has dual Routing Engines, remember to copy the script to the /var/db/scripts/event or /config/scripts/event directory on both Routing Engines. The commit synchronize command does not automatically copy scripts between Routing Engines.

3. Issue the **request system scripts event-scripts reload** operational mode command.

   ```
   user@host> request system scripts event-scripts reload
   ```

All event scripts are reloaded into the eventd process' memory.

Enabling an Event Script

Event scripts are stored on a device's hard drive in the /var/db/scripts/event directory or on the flash drive in the /config/scripts/event directory. Only users in the Junos OS **super-user** login class can access and edit files in these directories. For information about setting the storage location for scripts, see “Storing Event Scripts in Flash Memory” on page 451.

NOTE: If the device has dual Routing Engines and you want to enable an event script to execute on both Routing Engines, you must copy the script to the /var/db/scripts/event or /config/scripts/event directory on both Routing Engines. The commit synchronize command does not automatically copy scripts between Routing Engines.

You must enable an event op script before it can be executed. Include the **file *filename*** statement at the **[edit event-options events-script]** hierarchy level, specifying the name of an Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) file containing an event script. Only users who belong to the Junos **super-user** login class can enable event scripts.

```
[edit event-options event-script]
file filename;
```

The filename of an event script written in SLAX must include the **.slax** extension for the script to be enabled and executed. No particular filename extension is required for event scripts written in XSLT, but we strongly recommend that you append the **.xsl** extension.

To determine which event scripts are currently active on the device, either list the contents of the /var/run/scripts/event directory or use the **show** command to display the files included at the **[edit event-options event-script]** hierarchy level.

Configuring Checksum Hashes for an Event Script

You can configure one or more checksum hashes that can be used to verify the integrity of an event script before the script runs on the switch, router, or security device.

To configure a checksum hash:

1. Create the script.
2. Place the script in the /var/db/scripts/even directory on the device.
3. Run the script through one or more hash functions to calculate hash values.

   The Junos OS supports MD5, SHA-1, and SHA-256 hash functions.

   ```
   user@host>file checksum md5 /var/db/scripts/commit/script1.slax
     MD5 (/var/db/scripts/event/script1.slax) = 3af7884eb56e2d4489c2e49b26a39a97
   user@host>file checksum sha1 /var/db/scripts/commit/script1.slax
     SHA1 (/var/db/scripts/event/script1.slax) =
     00dc690fb08fb049577d012486c9a6dad34212c0
   user@host>file checksum sha-256 /var/db/scripts/commit/script1.slax
     SHA256 (/var/db/scripts/event/script1.slax) =
     150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
   ```

4. Configure the script.

   ```
   [edit event-options event-script]
   user@host#set file script1.slax checksum
     md5 3af7884eb56e2d4489c2e49b26a39a97
   [edit event-options event-script]
   user@host#set file script1.slax checksum
     sha-1 00dc690fb08fb049577d012486c9a6dad34212c0
   [edit event-options event-script]
   user@host#set file script1.slax checksum
     sha-256 150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
   ```

   During the execution of the script, the Junos OS recalculates the checksum value using the configured hash and verifies that the calculated value matches the configured value. If the values differ, the execution of the script fails. When you configure multiple checksum values with different hash algorithms, all the configured values must match the calculated values; otherwise, the script execution fails and the event policy fails.

Related Documentation

- Configuring Checksum Hashes for a Commit Script on page 192
- Configuring Checksum Hashes for an Op Script on page 319
- file checksum md5 command in the *System Basics and Services Command Reference*
- file checksum sha-256 command in the *System Basics and Services Command Reference*
- file checksum sha1 command in the *System Basics and Services Command Reference*

Executing an Event Script

When you issue the **commit** command, event scripts enabled at the **[edit event-options event-script]** hierarchy level are placed into system memory and enabled for execution. After the commit operation completes, an event script is executed in response to an event notification within an event policy. For more information, see "Executing Event Scripts in an Event Policy" on page 384.

Storing Event Scripts in Flash Memory

By default, event scripts are stored in the /var/db/scripts/event directory on the device's hard drive. To store them in flash memory instead, include the **load-scripts-from-flash** statement at the **[edit system scripts]** hierarchy level:

```
[edit system scripts]
load-scripts-from-flash;
```

The **load-scripts-from-flash** statement applies to all commit, operation, and event scripts. Event scripts are stored in the /config/scripts/event directory in flash memory. Changing the scripts' physical location has no effect on their operation.

NOTE: When you add or remove the load-scripts-from-flash statement in the configuration, you must manually move scripts from the hard drive to the flash drive, or vice versa, as appropriate. They are not moved automatically.

Specifying a Master Source for an Event Script

You can store a master copy of each event script in a central repository. This eases file management because you can make changes to the master event script in one place and then update the copy on each device where the event script is currently enabled.

To define the location of the master source file for an event script, include the **source** statement at the **[edit event-options event-script file *filename*]** hierarchy level:

```
[edit event-options event-script file filename]
source url;
```

- ***filename***—Name of the event script.
- ***url***—URL of the event script's master source file. Specify the source as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

The following example specifies an HTTP URL as the remote source for the iso.xsl file:

```
[edit event-options event-script]
file iso.xsl {
   source http://my.example.com/pub/scripts/iso.xsl;
}
```

Including the **source** statement in the configuration does not affect the local copy of the event script until you issue the **set refresh** command as described in "Updating an Event Script from the Master Source" on page 452. At that point, the master copy is retrieved from the specified URL and overwrites the local copy.

Updating an Event Script from the Master Source

To update a single event script from its master source, issue the **set refresh** command at the **[edit event-options event-script file *filename*]** hierarchy level. The master source must already be configured as described in "Specifying a Master Source for an Event Script" on page 451.

```
[edit event-options event-script file filename]
user@host# set refresh
```

To update all enabled event scripts from their master sources, issue the **set refresh** command at the **[edit event-options event-script]** hierarchy level:

```
[edit event-options event-script]
user@host# set refresh
```

When you issue the **set refresh** command, the switch, router, or security device immediately attempts to connect to the machine that houses the master source for the script files and retrieve a copy of each file. The master copy overwrites the script stored in the local event scripts directory. If a master source is not defined for a script, that script is not updated and a warning is issued.

The update operation occurs as soon as you issue the **set refresh** command. The **refresh** statement is not added to the configuration. In other words, for this statement the **set** command behaves like an operational mode command, instead of adding a statement to the configuration.

If a device has dual Routing Engines and you want the script to be updated on both Routing Engines, you must include the **refresh** statement in the configuration of both Routing Engines. The **commit synchronize** command does not cause the **refresh** statement to take effect on scripts in both Routing Engine directories.

Updating an Event Script from an Alternate Location

In addition to updating an event script from the master source defined by the **source** statement at the **[edit event-options event-script file *filename*]** hierarchy level, you also can update a script from an alternate location. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems. To update a single event script from the alternate source, issue the **set refresh-from** command at the **[edit event-options event-script file *filename*]** hierarchy level, specifying the location of the remote file:

```
[edit event-options event-script file filename]
user@host# set refresh-from url
```

To update all enabled event scripts from the alternate source, issue the **set refresh-from** command at the **[edit event-options event-script]** hierarchy level, specifying the location of the remote directory that houses the scripts:

```
[edit event-options event-script]
```

```
user@host# set refresh-from url
```

At both hierarchy levels:

url—URL of the remote event script or directory. Specify the source as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

When you issue the **set refresh-from** command, the switch, router, or security device immediately attempts to connect to the machine that houses the master source for the script files and retrieve a copy of each file. The master copy overwrites the script stored in the local event scripts directory. If a master source is not defined for a script, that script is not updated and a warning is issued.

The update operation occurs as soon as you issue the **set refresh-from** command. The **refresh-from** statement is not added to the configuration. In other words, for this statement, the **set** command behaves like an operational mode command, instead of adding a statement to the configuration.

If a device has dual Routing Engines and you want the script to be updated on both Routing Engines, you must issue the **set refresh-from** command on each Routing Engine separately. The **commit synchronize** command does not cause the **refresh-from** statement to update scripts on both Routing Engines.

Tracing Event Script Processing

Event script tracing operations track all event script operations and record them in a log file. The logged error descriptions provide detailed information to help you solve problems faster.

The default operation of event script tracing is to log important events in a file called escript.log located in the /var/log directory. When the file escript.log reaches 128 kilobytes (KB), it is renamed with a number 0 through 9 (in ascending order) appended to the end of the file and then compressed. The resulting files are escript.log.0.gz, then escript.log.1.gz, until there are 10 trace files. Then the oldest trace file (escript.log.9.gz) is overwritten. (For more information about how log files are created, see the *Junos OS System Log Messages Reference*.)

This section discusses the following topics:

Minimum Configuration for Enabling Traceoptions for Event Scripts

If no event script trace options are configured, the simplest way to view the trace output of an event script is to configure the **output** trace flag and issue the **show log escript.log | last** command. To do this, perform the following steps:

1. If you have not done so already, enable an event script by including the **file** statement at the **[edit event-options event-script]** hierarchy level:

   ```
   [edit event-options event-script]
   user@host# set file filename
   ```

2. Enable trace options by including the **traceoptions flag output** statement at the **[edit event-options event-script]** hierarchy level:

   ```
   [edit event-options event-script]
   user@host# set traceoptions flag output
   ```

3. Issue the **commit** command:

   ```
   [edit]
   user@host# commit
   ```

4. Display the resulting trace messages recorded in the */var/log/escript.log* file. At the end of the log is the output generated by the event script you enabled in Step 1 after a configured event policy is triggered and invokes the script. To display the end of the log, issue the **show log escript.log | last** operational mode command:

   ```
   [edit]
   user@host# run show log escript.log | last
   ```

Table 25 on page 454 summarizes useful filtering commands that display selected portions of the **escript.log** file.

Table 25: Event Script Tracing Operational Mode Commands

Task	Command
Display logging data associated with all event script processing.	**show log escript.log**
Display processing for only the most recent operation.	**show log escript.log | last**
Display processing for script errors.	**show log escript.log | match error**
Display processing for a particular script.	**show log escript.log | match *filename***

Example: Minimum Configuration for Enabling Traceoptions for Event Scripts

Display the trace output of the event script file **source-route.xsl**:

```
[edit]
event-options {
   event-script {
      file source-route.xsl;
      traceoptions flag output;
   }
}
```

```
[edit]
user@host# commit
[edit]
user@host# run show log escript.log | last
```

Configuring Tracing of Event Scripts

You cannot change the directory (/var/log) to which trace files are written. However, you can customize other trace file settings by including the following statements at the **[edit event-options event-script traceoptions]** hierarchy level:

```
[edit event-options event-script traceoptions]
file <filename> <files number> <size size> <world-readable | no-world-readable>;
flag all;
flag events;
flag input;
flag offline;
flag output;
flag rpc;
flag xslt;
no-remote-trace;
```

These statements are described in the following sections:

Configuring the Event Script Log Filename

By default, the name of the file that records trace output is escript.log. You can specify a different name by including the **file** statement at the **[edit event-options event-script traceoptions]** hierarchy level:

```
[edit event-options event-script traceoptions]
file filename;
```

Configuring the Number and Size of Event Script Log Files

By default, when the trace file reaches 128 KB in size, it is renamed and compressed to *filename*.0.gz, then *filename*.1.gz, and so on, until there are 10 trace files. Then the oldest trace file (*filename*.9.gz) is overwritten.

You can configure the limits on the number and size of trace files by including the following statements at the **[edit event-options event-script traceoptions file <*filename*>]** hierarchy level:

```
[edit event-options event-script traceoptions file <filename>]
files number size size;
```

For example, set the maximum file size to 640 KB and the maximum number of files to 20. When the file that receives the output of the tracing operation (*filename*) reaches 640 KB, it is renamed and compressed to *filename*.0.gz, and a new file called *filename* is created. When *filename* reaches 640 KB, *filename*.0.gz is renamed *filename*.1.gz and *filename* is renamed and compressed to *filename*.0.gz. This process repeats until there are 20 trace files. Then the oldest file (*filename*.19.gz) is overwritten.

The number of files can range from 2 through 1000 files. The file size can range from 10 KB through 1 gigabyte (GB).

NOTE:

If you set either a maximum file size or a maximum number of trace files, you also must specify the other parameter and a filename.

Configuring Access to Event Script Log Files

By default, access to the event script log file is restricted to the owner. You can manually configure access by including the **world-readable** or **no-world-readable** statement at the **[edit event-options event-script traceoptions file <*filename*>]** hierarchy level.

```
[edit event-options event-script traceoptions file <filename>]
(world-readable | no-world-readable);
```

The **no-world-readable** statement restricts event script log access to the owner. The **world-readable** statement enables unrestricted access to the event script log file.

Configuring the Event Script Trace Operations

By default, only important events are logged. You can configure the trace operations to be logged by including the following statements at the **[edit event-options event-script traceoptions]** hierarchy level:

```
[edit event-options event-script traceoptions]
flag all;
flag events;
flag input;
flag offline;
flag output;
flag rpc;
flag xslt;
```

Table 26 on page 456 describes the meaning of the event script tracing flags.

Table 26: Event Script Tracing Flags

Flag	Description	Default Setting
all	Trace all operations.	Off
events	Trace important events.	On
input	Trace event script input data.	Off
offline	Generate data for offline development.	Off
output	Trace event script output data.	Off
rpc	Trace event script RPCs.	Off

Table 26: Event Script Tracing Flags *(continued)*

Flag	Description	Default Setting
xslt	Trace the Extensible Stylesheet Language Transformations (XSLT) library.	Off

CHAPTER 29

Event Script Examples

This chapter provides the following sample event script:

Example: Limiting Event Script Output Based on a Specific Event Type

In situations where an event policy is triggered by multiple event types, you can limit the number of events that trigger the event script. For example, the following event policy triggers the **event-details.slax** event script whenever a **ui_login_event** or **ui_logout_event** occurs.

```
event-options {
  policy event-detail {
    events [ ui_login_event ui_logout_event ];
    then {
      event-script event-details.slax {
        output-filename systemlog;
        destination /tmp;
      }
    }
  }
}
```

The **event-details.slax** event script writes a log file only when the **ui_login_event** event occurs.

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns ext = "http://xmlsoft.org/XSLT/namespace";

var $event-definition = {
  <event-options> {
    <policy> {
      <namex> "event-detail";
      <eventsx> "ui_login_event";
      <thenx> {
        <event-scriptx> {
          <namex> "event_detail.slax";
          <output-filenamex> "foo";
```

```
            <destinationx> {
              <namex> "foo";
            }
          }
        }
      }
    }
  }

match / {
  <event-script-resultsx> {
    <event-triggered-this-policyx> {
      expr event-script-input/trigger-event/id;
    }
    <type-of-eventx> {
      expr event-script-input/trigger-event/type;
    }
    <process-namex> {
      expr event-script-input/trigger-event/attribute-list/attribute/name;
    }
  }
}
```

CHAPTER 30

Summary of Event Script Configuration Statements

This chapter describes each configuration statement for event scripts. The statements are organized alphabetically.

checksum

Syntax checksum (md5 | sha-256 | sha1) *hash*;

Hierarchy Level [edit event-options **event-script file** *filename*],
[edit system **scripts commit file** *filename*],
[edit system **scripts op file** *filename*]

Release Information Statement introduced in Junos OS Release 9.5.

Description For Junos OS commit scripts and op scripts, specify the MD5, SHA-1, or SHA-256 checksum hash. When it executes a local event, commit, or op script, the Junos OS verifies the authenticity of the script by using the configured checksum hash.

Options **md5** ***hash***—MD5 checksum of this script.

sha-256 ***hash***—SHA-256 checksum of this script.

sha1 ***hash***—SHA-1 checksum of this script.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring Checksum Hashes for a Commit Script on page 192
- Configuring Checksum Hashes for an Event Script on page 450
- Configuring Checksum Hashes for an Op Script on page 319
- Executing an Op Script from a Remote Site on page 321
- file checksum md5 command in the *System Basics and Services Command Reference*
- file checksum sha-256 command in the *System Basics and Services Command Reference*
- file checksum sha1 command in the *System Basics and Services Command Reference*

event-script

Syntax

```
event-script {
  file filename {
    checksum (md5 | sha-256 | sha1) hash;
    refresh;
    refresh-from url;
    remote-execution {
      remote-hostname {
        passphrase user-password;
        username user-login;
      }
    }
    source url;
  }
  refresh;
  refresh-from url;
  traceoptions {
    file <filename> <files number> <size size> <world-readable | no-world-readable>;
    flag flag;
    no-remote-trace;
  }
}
```

Hierarchy Level [edit event-options]

Release Information Statement introduced in Junos OS Release 7.6.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description For Junos OS event scripts, configure scripting mechanisms.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Implementing Event Scripts on page 448

file

Syntax

```
file filename {
  checksum (md5 | sha-256 | sha1) hash;
  refresh;
  refresh-from url;
  remote-execution {
    remote-hostname {
      passphrase user-password;
      username user-login;
    }
  }
  source url;
}
```

Hierarchy Level [edit event-options event-script]

Release Information Statement introduced in Junos OS Release 7.6.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description For Junos OS event scripts, enable an event script that is located in the **/var/db/scripts/event** directory.

Options ***filename***—The name of an Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) file containing an event script.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Enabling an Event Script on page 449

refresh

Syntax refresh;

Hierarchy Level [edit **event-options event-script**],
[edit **event-options event-script file** *filename*]

Release Information Statement introduced in Junos OS Release 9.6.
Statement introduced in Junos OS Release 9.6 for EX Series switches.

Description For Junos OS event scripts, overwrite the local copy of all enabled event scripts or a single enabled script located in the **/var/db/scripts/event** directory with the copy located at the source URL, specified in the **source** statement at the same hierarchy level.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Updating an Event Script from the Master Source on page 452
- **refresh-from (Event Scripts) on page 464**
- **source on page 466**

refresh-from (Event Scripts)

Syntax refresh-from *url*;

Hierarchy Level [edit **event-options event-script**],
[edit **event-options event-script file** *filename*]

Release Information Statement introduced in Junos OS Release 9.6.
Statement introduced in Junos OS Release 9.6 for EX Series switches.

Description For Junos OS event scripts, overwrite the local copy of all enabled event scripts or a single enabled script located in the **/var/db/scripts/event** directory with the copy located at a URL other than the URL specified in the **source** statement.

Options *url*—Source specified as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Updating an Event Script from an Alternate Location on page 452
- **refresh on page 464**
- **source on page 466**

remote-execution

Syntax

```
remote-execution {
  remote-hostname {
    passphrase user-password;
    username user-login;
  }
}
```

Hierarchy Level [edit **event-options event-script file** *filename*]

Release Information Statement introduced in Junos OS Release 9.6.
Statement introduced in Junos OS Release 9.6 for EX Series switches.

Description For Junos OS event scripts, enable event scripts to invoke RPCs on a local or remote host.

Options

passphrase ***user-password***—User's password for the remote host.

remote-hostname—Name of the remote host with which the event script will communicate.

username ***username***—User's login name for the remote host.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Using RPCs and Operational Mode Commands in Event Scripts on page 441

source

Syntax source *url*;

Hierarchy Level [edit **event-options event-script file** *filename*]

Release Information Statement introduced in Junos OS Release 9.6.
Statement introduced in Junos OS Release 9.6 for EX Series switches.

Description For Junos OS event scripts, specify the location of the source file for an enabled script located in the **/var/db/scripts/event** directory. When you include the **refresh** statement at the same hierarchy level, the local copy is overwritten by the version stored at the specified URL.

Options *url*—Master source file for an event script specified as an HTTP URL, FTP URL, or scp-style remote file specification.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- **refresh on page 464**
- **refresh-from (Event Scripts) on page 464**
- Specifying a Master Source for an Event Script on page 451

traceoptions (Event Scripts)

Syntax

```
traceoptions {
  file <filename> <files number> <size size> <world-readable | no-world-readable>;
  flag flag;
  no-remote-trace;
}
```

Hierarchy Level [edit **event-options event-script**]

Release Information Statement introduced in Junos OS Release 7.6.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description Define tracing operations for event scripts.

Default If you do not include this statement, no event script–specific tracing operations are performed.

Options *filename*—Name of the file to receive the output of the tracing operation. All files are placed in the directory /var/log. By default, event script process tracing output is placed in the file **escript.log**. If you include the **file** statement, you must specify a filename. To retain the default, you can specify escript.log as the filename.

files ***number***—(Optional) Maximum number of trace files. When a trace file named *trace-file* reaches its maximum size, it is renamed and compressed to *trace-file*.0.gz. When *trace-file* again reaches its maximum size, *trace-file*.0.gz is renamed *trace-file*.1.gz and *trace-file* is renamed and compressed to *trace-file*.0.gz. This renaming scheme continues until the maximum number of trace files is reached. Then the oldest trace file is overwritten.

If you specify a maximum number of files, you also must specify a maximum file size with the **size** option and a filename.

Range: 2 through 1000

Default: 10 files

flag—Tracing operation to perform. To specify more than one tracing operation, include multiple **flag** statements. You can include the following flags:

- **all**—Log all operations
- **events**—Log important events
- **input**—Log event script input data
- **offline**—Generate data for offline development
- **output**—Log event script output data
- **rpc**—Log event script RPCs
- **xslt**—Log the XSLT library

no-world-readable—Restrict file access to owner. This is the default.

size *size*—(Optional) Maximum size of each trace file, in kilobytes (KB), megabytes (MB), or gigabytes (GB). When a trace file named ***trace-file*** reaches this size, it is renamed and compressed to ***trace-file*.0.gz**. When ***trace-file*** again reaches its maximum size, ***trace-file*.0.gz** is renamed ***trace-file*.1.gz** and ***trace-file*** is renamed and compressed to ***trace-file*.0.gz**. This renaming scheme continues until the maximum number of trace files is reached. Then the oldest trace file is overwritten.

If you specify a maximum file size, you also must specify a maximum number of trace files with the **files** option and a filename.

Syntax: ***x*k** to specify KB, ***x*m** to specify MB, or ***x*g** to specify GB

Range: 10 KB through 1 GB

Default: 128 KB

world-readable—Enable unrestricted file access.

Required Privilege Level

maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Tracing Event Script Processing on page 453

PART 6

Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

S

T

Index of Statements and Commands