Juniper Networks | Engineering Simplicity

Junos® OS

Automation Scripting User Guide

Published
2024-12-11

JUNOS

Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

Junos® OS Automation Scripting User Guide

The information in this document is current as of the date on the title page.

YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. 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

Table of Contents

8 Commit Scripts

About This Guide

Use this guide to develop on-box Junos OS automation scripts in Python, SLAX, or XSLT to automate operational and configuration tasks on Junos devices or to create macros to simplify complex configurations. Junos OS automation scripts include commit scripts and macros, operation (op) scripts, event policies and event scripts, and SNMP scripts.

RELATED DOCUMENTATION

- Juniper Techwiki: Automation Scripting
- Day One: Navigating the Junos XML Hierarchy
- This Week: Mastering Junos Automation Programming
- This Week: Applying Junos Automation
- This Week: Junos Automation Reference for SLAX 1.0

1

PART

Overview

CHAPTER 1

Junos Automation Scripts Overview

IN THIS CHAPTER

Junos Automation Scripts Overview

IN THIS SECTION

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

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 the following scripting languages:

- Extensible Stylesheet Language Transformations (*XSLT*)
- Stylesheet Language Alternative syntaX (SLAX)
- Python

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

This topic describes the different types of functionality for Junos automation.

Junos Configuration Automation: Commit Scripts

Junos configuration automation uses commit scripts to automate the commit process. 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 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. Op scripts can be executed manually in the CLI or upon user login, or they can be called from another script. Op scripts can process user arguments and 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
- Monitor the overall status of a device by periodically checking network warning parameters, such as high CPU usage.

Junos Event Automation: Event Scripts and Event Policy

Junos event automation uses event policies and event scripts to instruct 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 operational mode commands
- Execute event scripts
- Modify the configuration

Event Scripts

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

- Automatically diagnosing and fixing problems in the network
- Monitoring the overall status of a device
- Running automatically as part of an event policy that detects periodic error conditions
- Changing the configuration in response to a problem

Junos SNMP Automation: SNMP Scripts

SNMP scripts provide the flexibility to support custom MIBs. SNMP scripts are triggered automatically when the SNMP manager requests information from the SNMP agent for an object identifier (OID) that is mapped to an SNMP script for an unsupported OID. The script acts like an SNMP subagent, and the system sends the return value from the script to the network management system (NMS).

You can map an SNMP script to one or more OIDs using the `oid` statement at the `[edit system scripts snmp file script-name]` hierarchy level.

RELATED DOCUMENTATION

Overview of Junos Automation Enhancements on Devices Running Junos OS with Enhanced Automation

IN THIS SECTION

The Junos Automation Enhancements are designed to support the increasing needs of large data centers for more automation and programmability.

Features of the Junos Automation Enhancements

To use the Junos Automation Enhancements, you must install the software bundle that contains Enhanced Automation. The filename of the software bundle varies, depending on the switch. For example, for the QFX5200 switch, you would install an image with the filename `jinstall-qfx-5e-flex-`*`version`*`-domestic-signed.tgz`. This software bundle is identical to the other software bundle except that

Veriexec is disabled, which enables you to run unsigned programs, such as programs that you develop with Python, Chef, and Puppet. The Junos Automation Enhancements include the following features:

- The factory default configuration is a Layer 3 configuration. (The standard default factory configuration on some device series is Layer 2.)
- Safeguards ensure that you cannot overwrite essential Junos OS files, including system log notifications.
- The installation automatically sets up and reserves a 1 GB user partition on your system. You can use this partition to store your binaries and additional packages.
- The user partition is not overwritten when you upgrade or downgrade the software to a OS image that does not contain the automation enhancements.

  NOTE: If you make changes to the user partition while performing a unified in-service software upgrade (unified ISSU), the changes might be lost.

- The Python interpreter is included by default.
  - You can invoke Python directly from the shell. See "How to Use Python Interactive Mode on Devices Running Junos OS" on page 289.
  - Starting with Junos OS Release 14.1X53-D10, three open-source Python modules are pre-installed in the **jinstall-qfx-5-flex-x.tgz** software bundle.
- Chef for Junos OS and Puppet for Junos OS automation tools for provisioning and managing computer networking and storage resources are included.
  - For further information on Chef, see Chef for Junos Getting Started Guide.
  - For further information on Puppet, see Puppet for Junos OS Documentation.

  NOTE: For full compatibility, you must use only Chef for Junos OS and Puppet for Junos OS rather than the standard FreeBSD versions of Chef and Puppet software.

CAUTION: Download additional third party packages at your own risk.

RELATED DOCUMENTATION

How to Use Python Interactive Mode on Devices Running Junos OS | 289

Understanding Automation Script Support on the QFabric System Director Devices

Junos OS automation consists of a suite of tools used to automate operational and configuration tasks on network devices running Junos OS. The 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.

> **NOTE**: Event policies and event scripts are not supported on the QFabric system at this time.

The QFabric system supports Junos OS automation scripts that are written in Stylesheet Language Alternative Syntax (SLAX) version 1.0.

Commit scripts automate the commit process and enforce custom configuration rules. You can use commit scripts to generate specific errors and warnings, and customize configurations and configuration templates. When a candidate configuration is committed, it is inspected by each active commit script. If a configuration violates your custom rules and the scripts generate an error, the commit fails. If the commit is successful, any configuration changes (both transient and permanent) are incorporated into the active configuration before it is passed to the Director software, which distributes the configuration to all applicable QFabric system components, including Node devices and Node servers.

Op scripts automate operational and troubleshooting tasks. Op scripts can be executed manually from the Junos OS CLI or NETCONF XML management protocol, or they can be called from another script.

The QFabric system supports the following automation script features:

- Commit scripts and op scripts
- Scripts written in SLAX version 1.0

The QFabric system has the following requirements when using automation scripts:

- Scripts are configured and deployed from the Director group. Since there is more than one Director device in a Director group, scripts must be deployed by each Director device or deployed in the shared media space.
- Scripts are stored in the shared media at this location: **/pbdata/mgd_shared/*partition-ip*/var/db/scripts**. Under this directory, commit scripts are stored in the **commit** subdirectory, and op scripts are stored in the **op** subdirectory.
- Scripts are not stored in flash memory.

RELATED DOCUMENTATION

How Commit Scripts Work | 533
How Op Scripts Work | 821
Required Boilerplate for Commit Scripts | 542
Required Boilerplate for Op Scripts | 822
Control the Execution of Commit Scripts in the QFabric System | 561

CHAPTER 2

Junos XML Management Protocol and Junos XML API Overview

IN THIS CHAPTER

XML Overview

IN THIS SECTION

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. XML tags look much like 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* or *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 an 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 element. This is an XML convention, and the brackets are a required part of the complete element name. They are not to be confused with the angle brackets used in the Juniper Networks documentation to indicate optional parts of Junos OS CLI command strings.

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

When discussing XML elements in text, this documentation conventionally uses just the opening tag to represent the complete 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>` 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` 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, event, and SNMP 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. A file called a *document type definition*, or *DTD*, defines these rules. The rules define the contexts in which each tagged item can—and in some cases must—occur. A DTD:

- Lists every element that can appear in the document or data set
- Defines the parent-child relationships between the tags
- Specifies other tag characteristics

The same DTD can apply to many XML documents or data sets.

RELATED DOCUMENTATION

Junos XML Management Protocol and Junos XML API Overview | 14
XML and Junos OS Overview | 12

XML and Junos OS Overview

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, 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 OS configuration statements and operational mode commands. It defines an XML equivalent for all statements in the Junos OS 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.

To display the configuration or operational mode command output as Junos XML tag elements instead of as the default formatted ASCII, issue the command, and pipe the output to the `display xml` command. Infrastructure tag elements in the response belong to the Junos XML management protocol. The tag elements that describe Junos OS configuration or operational data belong to the Junos XML API, which defines the Junos OS content that can be retrieved and manipulated by both the Junos XML management protocol and the NETCONF XML management protocol operations. The following example compares the text and XML 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/10.4R1/junos">
    <alarm-information xmlns="http://xml.juniper.net/junos/10.4R1/junos-alarm">
        <alarm-summary>
            <no-active-alarms/>
        </alarm-summary>
    </alarm-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

To display the Junos XML API representation of any operational mode command, issue the command, and pipe the output to the `display xml rpc` command. The following example shows the Junos XML API request tag for the `show chassis alarms` command.

```
user@host> show chassis alarms | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4R1/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 Junos XML API request tag that is sent to 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 the Junos XML API `<get-alarm-information>` request tag and sends the XML request to the Junos OS infrastructure for processing. 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 OS automation scripts use XML to communicate with the host device. Junos OS provides XML-formatted input to a script. The script processes the input source tree and then returns XML-formatted output to 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 Junos OS for processing. Commit script input consists of an XML representation of the post-inheritance candidate configuration file. Event scripts receive an XML document containing the description of the triggering event. All script input documents contain information pertaining to the Junos OS environment, and some scripts receive additional script-specific input that depends on the script type.

RELATED DOCUMENTATION

Junos XML API Explorer

Junos XML Management Protocol and Junos XML API Overview

The Junos XML Management Protocol is an Extensible Markup Language (XML)-based protocol that client applications use to manage the configuration on Junos devices. It uses an XML-based data encoding for the configuration data and remote procedure calls (RPCs). The Junos XML protocol defines basic operations that are equivalent to configuration mode commands in the 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 information and change the configuration on a switch, router, or security 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 operating system 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 protocol and Junos XML API 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. 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 Junos devices 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 use 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

IN THIS SECTION

The Junos XML management protocol and Junos XML *API* fully document all options for every supported Junos OS operational request, all statements in the Junos OS 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 Management Protocol and Junos XML API Overview | 14

XML Overview | 9

2

PART

Automation Scripting Using XSLT

CHAPTER 3

XSLT Overview

IN THIS CHAPTER

XSLT Overview

IN THIS SECTION

Commit scripts, op scripts, event scripts, and SNMP 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 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 Junos OS can easily express in XML) and for generating errors and warnings (which 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.

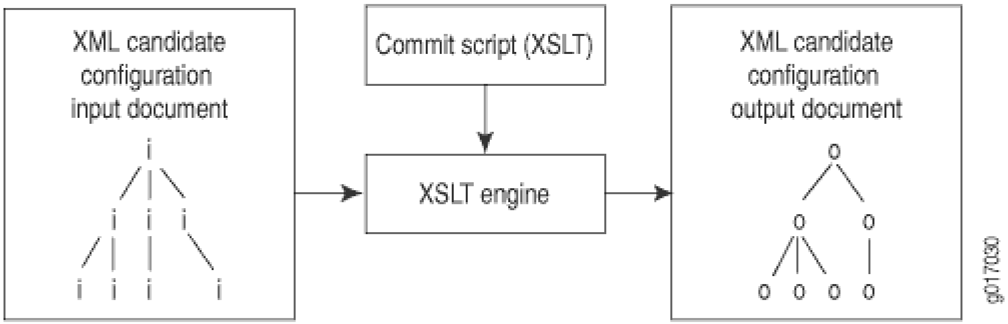
Although XSLT provides a powerful scripting ability, its focus is specific and limited. It does not make Junos OS vulnerable to arbitrary or malicious programmers. XSLT restricts programmers from performing haphazard operations, such as opening random Transmission Control Protocol (TCP) ports, forking 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, event scripts, and SNMP 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, event scripts, and SNMP scripts, the XSLT engine is a function of the Junos OS management process (*mgd*).

Figure 1 on page 21 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 1 on page 21.

Table 1: 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
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 | 22
XSLT Context (Dot) Overview | 39

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 style sheet 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 9.

XPath Overview

IN THIS SECTION

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, event scripts, and SNMP 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 542.

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, consult a comprehensive XPath reference guide. XPath is fully described in the W3C specification at http://w3c.org/TR/xpath.

XSLT Templates Overview

IN THIS SECTION

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

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 <div class="author"> 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" on page 72.

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 28.

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: Require and Restrict Configuration Statements" on page 790.

RELATED DOCUMENTATION

XSLT Parameters Overview

IN THIS SECTION

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 OS at script initialization must be defined as a global parameter. Global parameter declarations are placed just after the style sheet declarations. A script can assign a default value to the global parameter, which is used in the event that Junos OS 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 Variables Overview

In XSLT scripts, 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 style sheet. 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.

```
<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>
```

The resulting system log message is as follows:

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

Table 2 on page 33 provides examples of XSLT variable declarations along with pseudocode explanations.

Table 2: 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`.

Table 2: Examples and Pseudocode for XSLT Variable Declaration *(Continued)*

Variable Declaration	Pseudocode Explanation
`<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 "Create a Commit Script Macro to Read the Custom Syntax and Generate Related Configuration Statements" on page 641.

RELATED DOCUMENTATION

xsl:variable | 77

XSLT Programming Instructions Overview

IN THIS SECTION

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, event, and SNMP scripts, which provide flow control within a script, are described in the following sections:

<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 one is present, 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 3 on page 37 presents examples that use several XSLT programming instructions along with pseudocode explanations.

Table 3: Examples and Pseudocode for XSLT Programming Instructions

<table>
<tr><th>Programming Instruction</th><th>Pseudocode Explanation</th></tr>
<tr><td><pre>
<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>
</pre></td><td>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.</td></tr>
<tr><td><pre>
<xsl:for-each select="interfaces/
interface[starts-with(name, 'ge-')]/unit">
</pre></td><td>For each Gigabit Ethernet interface configured at the [edit interfaces ge-fpc/pic/port unit logical-unit-number] hierarchy level.</td></tr>
<tr><td><pre>
<xsl:for-each select="data[not(value)]/name">
</pre></td><td>Select any macro parameter that does not contain a parameter value.

In other words, match all apply-macro statements of the following form:<pre>
apply-macro apply-macro-name {
    parameter-name;
}
</pre>And ignore all apply-macro statements of the form:<pre>
apply-macro apply-macro-name {
    parameter-name parameter-value;
}
</pre></td></tr>
</table>

Table 3: Examples and Pseudocode for XSLT Programming Instructions *(Continued)*

Programming Instruction	Pseudocode Explanation
`<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.

RELATED DOCUMENTATION

xsl:choose | 57
xsl:for-each | 62
xsl:if | 63
xsl:otherwise | 66
xsl:when | 78

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 4

Standard XPath and XSLT Functions Used in Automation Scripts

IN THIS CHAPTER

concat()

IN THIS SECTION

Syntax

```
string concat(string, string+)
```

Description

Return the concatenation of the arguments.

Usage Examples

See "Example: Limit the Number of E1 Interfaces" on page 742, "Example: Control IS-IS and MPLS Interfaces" on page 710, "Example: Adding T1 Interfaces to a RIP Group" on page 663, "Example: Configure Administrative Groups for LSPs" on page 684, and "Example: Configure Dual Routing Engines" on page 696.

RELATED DOCUMENTATION

contains()

IN THIS SECTION

Syntax

```
boolean contains(string, string)
```

Description

Return TRUE if the first string argument contains the second string argument, otherwise return FALSE.

Usage Examples

See "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

RELATED DOCUMENTATION

count()

IN THIS SECTION

Syntax

```
number count(node-set)
```

Description

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

Usage Examples

See "Example: Limit the Number of E1 Interfaces" on page 742.

RELATED DOCUMENTATION

last()

IN THIS SECTION

Syntax

```
number last()
```

Description

Return the index of the last node in the list that is currently being evaluated, which is equal to the number of items in the processed node list.

Usage Examples

See "Example: Limit the Number of E1 Interfaces" on page 742.

RELATED DOCUMENTATION

name()

IN THIS SECTION

Syntax

```
string name(<node-set>)
```

Description

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

Usage Examples

See "emit-change Template (SLAX and XSLT) and emit_change (Python) " on page 434.

RELATED DOCUMENTATION

count() | 43

last() | 44

not() | 46

position() | 47

not()

IN THIS SECTION

Syntax

```
boolean not(boolean)
```

Description

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

Usage Examples

See "Example: Require and Restrict Configuration Statements" on page 790, "Example: Control IS-IS and MPLS Interfaces" on page 710, "Example: Configure a Default Encapsulation Type" on page 691, "Example: Control LDP Configuration" on page 716, "Example: Adding a Final then accept Term to a Firewall" on page 657, "Example: Configure Administrative Groups for LSPs" on page 684, "Example: Configure Dual Routing Engines" on page 696, and "Example: Prevent Import of the Full Routing Table" on page 781.

RELATED DOCUMENTATION

position()

IN THIS SECTION

Syntax

```
number position()
```

Description

Return the current context position among the list of nodes that are currently being evaluated. The context position is the index of the node within the node-set being evaluated by a predicate, or if `position()` is being used outside of a predicate, then it is the index of the current node within the current node list. The initial position is 1 and the final position is equal to the context size, which can be retrieved through the last() function.

Usage Examples

The following op script shows the effect of using the `position()` function in both location path predicates as well as within for-each loops.

```
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 / {
    var $host-name-set := {
        <host-name> "PE1";
        <host-name> "P1";
        <host-name> "P2";
        <host-name> "PE2";
    }

    var $first-host-name = $host-name-set/host-name[ position() == 1 ];
    expr jcs:output( "First host-name: ", $first-host-name );

    var $first-p-host-name = $host-name-set/host-name[not(starts-with(.,"PE"))][position() == 1];

    expr jcs:output( "First P host-name: ", $first-p-host-name );

    expr jcs:output( "All host-names:" );
    for-each( $host-name-set/host-name ) {
        expr jcs:output( position(), ": ", . );
    }

    expr jcs:output( "P host-names only:" );
    for-each( $host-name-set/host-name[ not(starts-with( ., "PE" ))] ) {
        expr jcs:output( position(), ": ", . );
    }
}
```

```
user@host> op position
First host-name: PE1
First P host-name: P1
All host-names:
1: PE1
2: P1
3: P2
4: PE2
P host-names only:
```

```
1: P1
2: P2
```

RELATED DOCUMENTATION

starts-with()

IN THIS SECTION

Syntax

```
boolean starts-with(string, string)
```

Description

Return TRUE if the first string argument starts with the second string argument, otherwise return FALSE.

Usage Examples

See "Example: Impose a Minimum MTU Setting" on page 732, "Example: Limit the Number of E1 Interfaces" on page 742, "Example: Limit the Number of ATM Virtual Circuits" on page 736, "Example:

Adding T1 Interfaces to a RIP Group" on page 663, "Example: Configure a Default Encapsulation Type" on page 691, and "Example: Configure Dual Routing Engines" on page 696.

RELATED DOCUMENTATION

string-length()

IN THIS SECTION

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 Configure Logical Interfaces and IP Addresses" on page 674.

RELATED DOCUMENTATION

substring-after()

IN THIS SECTION

Syntax

```
string substring-after(string, string)
```

Description

Return the portion of the first string argument that follows the occurrence of the second argument substring within the first. If the second string is not contained in the first string, or if the second string is empty, the function returns an empty string.

Usage Examples

See "Example: Limit the Number of E1 Interfaces" on page 742 and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

RELATED DOCUMENTATION

substring-before()

IN THIS SECTION

Syntax

```
string substring-before(string, string)
```

Description

Return the portion of the first string argument that precedes the occurrence of the second argument substring within the first. If the second string is not contained in the first string, or if the second string is empty, the function returns an empty string.

Usage Examples

See "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

RELATED DOCUMENTATION

concat() | 41

contains() | 42

starts-with() | 49

string-length() | 50

substring-after() | 51

CHAPTER 5

Standard XSLT Elements and Attributes Used in Automation Scripts

IN THIS CHAPTER

xsl:apply-templates

IN THIS SECTION

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. If the `select` attribute is not included, the script recursively processes all child nodes of the current node. If the `select` attribute is present, the processor only applies templates to the child elements that match the expression of the `select` attribute, which must evaluate to a node-set. 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 (Optional) 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 657 and "Example: Prevent Import of the Full Routing Table" on page 781.

RELATED DOCUMENTATION

xsl:call-template

IN THIS SECTION

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 define the parameters that are passed to the template.

Attributes

name Specifies the name of the template to call.

Usage Examples

See "Example: Require and Restrict Configuration Statements" on page 790, "Example: Impose a Minimum MTU Setting" on page 732, and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

RELATED DOCUMENTATION

xsl:choose

IN THIS SECTION

Syntax

```
<xsl:choose>
    <xsl:when test="boolean-expression">
    ...
```

```
    </xsl:when>
    <xsl:otherwise>
     ...
    </xsl:otherwise>
</xsl:call-template>
```

Description

Evaluate multiple conditional tests, and execute instructions for the first test that evaluates to TRUE or execute an optional default set of instructions if all tests evaluate to FALSE. The `<xsl:choose>` instruction contains one or more `<xsl:when>` elements, each of which tests a Boolean expression. If the test evaluates to TRUE, the *XSLT* processor executes the instructions in the `<xsl:when>` element, and ignores all subsequent `<xsl:when>` elements. 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 one is present, is processed.

Usage Examples

See "Example: Configure Dual Routing Engines" on page 696, "Example: Prevent Import of the Full Routing Table" on page 781, and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

RELATED DOCUMENTATION

xsl:comment

IN THIS SECTION

Syntax

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

Description

Generate a comment node 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 comments delimited by `<!--` and `-->` such as `<!-- ... Insert your comment here ... -->`, 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 657.

RELATED DOCUMENTATION

xsl:import | 64

xsl:stylesheet | 68

xsl:copy-of

IN THIS SECTION

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	XPath expression specifying which nodes to copy.

Usage Examples

See "Example: Require and Restrict Configuration Statements" on page 790.

RELATED DOCUMENTATION

xsl:element | 61

xsl:text | 75

xsl:value-of | 76

xsl:element

IN THIS SECTION

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 and evaluated at run time. To do this, enclose an XML element in curly brackets, as in `<xsl:element name="{$isis-level-1}"`.

Usage Examples

See "Example: Create a Complex Configuration Based on a Simple Interface Configuration" on page 722.

RELATED DOCUMENTATION

xsl:copy-of | 60

xsl:text | 75

xsl:value-of | 76

xsl:for-each

IN THIS SECTION

Syntax

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

Description

Include a looping mechanism that repeats XSL processing for each XML element in the specified node-set. The element nodes are selected by the *XPath* expression defined by the `select` attribute. Each of the nodes is then processed according to the instructions contained in the `<xsl:for-each>` element.

Attributes

select Specifies an XPath expression that selects the nodes to be processed.

Usage Examples

See "Example: Require and Restrict Configuration Statements" on page 790, "Example: Impose a Minimum MTU Setting" on page 732, "Example: Limit the Number of E1 Interfaces" on page 742, "Example: Adding T1 Interfaces to a RIP Group" on page 663, "Example: Configure Administrative Groups for LSPs" on page 684, and "Example: Configure Dual Routing Engines" on page 696.

RELATED DOCUMENTATION

xsl:if

IN THIS SECTION

Syntax

```
<xsl:if test="expression">
    ...
</xsl:if>
```

Description

Include a conditional construct that causes instructions to be processed if the expression held in the `test` attribute evaluates to `TRUE`.

Attributes

test	Specifies the expression to evaluate.

Usage Examples

- "Example: Adding T1 Interfaces to a RIP Group" on page 663
- "Example: Configure Dual Routing Engines" on page 696
- "Example: Limit the Number of E1 Interfaces" on page 742
- "Example: Require and Restrict Configuration Statements" on page 790

RELATED DOCUMENTATION

XSLT Programming Instructions Overview | 34

xsl:choose | 57

xsl:for-each | 62

xsl:otherwise | 66

xsl:when | 78

xsl:import

IN THIS SECTION

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 "Example: Adding a Final then accept Term to a Firewall" on page 657, "Example: Configure a Default Encapsulation Type" on page 691, "Example: Configure Dual Routing Engines" on page 696, "Example: Control IS-IS and MPLS Interfaces" on page 710, "Example: Prepend a Global Policy" on page 774, and "Example: Prevent Import of the Full Routing Table" on page 781.

RELATED DOCUMENTATION

Understanding Named Templates in Junos OS Automation Scripts | 427
Global Parameters and Variables in Junos OS Automation Scripts | 327
Required Boilerplate for Commit Scripts | 542
Required Boilerplate for Event Scripts | 1084
Required Boilerplate for Op Scripts | 822
xsl:stylesheet | 68

xsl:otherwise

IN THIS SECTION

Syntax

```
<xsl:otherwise>
    ...
</xsl:otherwise>
```

Description

Within an `<xsl:choose>` instruction, include a default set of instructions that are processed if none of the expressions defined in the `test` attributes of the `<xsl:when>` elements evaluate to `TRUE`.

Usage Examples

See "Example: Configure Dual Routing Engines" on page 696 and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

RELATED DOCUMENTATION

xsl:param

IN THIS SECTION

Syntax

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

Description

Declare a parameter for a template or for the style sheet as a whole. A template parameter must be declared within the template element. A global parameter, the scope of which is the entire style sheet, must be declared at the top level of the style sheet.

Attributes

name Defines the name of the parameter.

select (Optional) *XPath* expression defining 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 content; we recommend using the `select` attribute so as not to create a result tree fragment.

Usage Examples

- "Example: Impose a Minimum MTU Setting" on page 732

- "Example: Limit the Number of ATM Virtual Circuits" on page 736
- "Example: Limit the Number of E1 Interfaces" on page 742
- "Example: Prevent Import of the Full Routing Table" on page 781
- "Example: Require and Restrict Configuration Statements" on page 790

RELATED DOCUMENTATION

XSLT Parameters Overview | 28

XSLT Templates Overview | 25

xsl:apply-templates | 55

xsl:call-template | 56

xsl:template | 70

xsl:variable | 77

xsl:with-param | 80

xsl:stylesheet

IN THIS SECTION

Syntax

```
<xsl:stylesheet version="1.0" xmlns:ext="URI">
    <xsl:import href="../import/junos.xsl"/>
    ...
</xsl:stylesheet>
```

Description

Include the document element for the style sheet. This element defines the root element of the style sheet, which contains all the top-level elements such as global variable and parameter declarations, import elements, and templates. Optionally, namespace mappings, which include an extension prefix and *Uniform Resource Identifier* (URI), can be included as attributes in the opening `<xsl:stylesheet>` tag.

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 Uniform Resource Identifier (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. Junos OS supports XSLT version 1.0.
xmlns:*ext*="*URI*	(Optional) Maps a namespace prefix to the URI for extension elements.

Usage Examples

- "Example: Adding a Final then accept Term to a Firewall" on page 657
- "Example: Configure Administrative Groups for LSPs" on page 684
- "Example: Configure a Default Encapsulation Type" on page 691
- "Example: Customize Output of the show interfaces terse Command Using an Op Script" on page 877

RELATED DOCUMENTATION

Required Boilerplate for Commit Scripts | **542**

Required Boilerplate for Event Scripts | **1084**

Required Boilerplate for Op Scripts | **822**

XSLT Namespace | **22**

xsl:import | **64**

xsl:template

IN THIS SECTION

Syntax

```
<xsl:template match="pattern" mode="qualified-name" name="qualified-name priority="integer">
    <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 an XML document. For example, `match="/"` matches the root element of the document. Although the `match` and `name` attributes are optional, one of the two attributes must be included in the template definition.

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. If more than one template matches a node in the specified mode, the priority attribute 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` attribute.

You can pass template parameters 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 (Optional) *XPath* expression specifying the nodes to which to apply the template. If this attribute is omitted, the `name` attribute must be included.

mode (Optional) Indicate the mode in which a template needs to be applied for the template to be used.

name (Optional) Specify a name for the template. Named templates can be explicitly called with the `<xsl:call-template>` element. If the `name` attribute is omitted, the `match` attribute must be included.

priority (Optional) Specify a numeric priority for the template.

Usage Examples

- "Example: Adding a Final then accept Term to a Firewall" on page 657
- "Example: Adding T1 Interfaces to a RIP Group" on page 663
- "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674
- "Example: Customize Output of the show interfaces terse Command Using an Op Script" on page 877
- "Example: Require and Restrict Configuration Statements" on page 790

RELATED DOCUMENTATION

XSLT Templates Overview | **25**

XSLT Parameters Overview | **28**

xsl:apply-templates | **55**

xsl:call-template | **56**

xsl:param | **67**

xsl:variable | **77**

xsl:with-param | **80**

xsl:template match="/" Template

IN THIS SECTION

Syntax

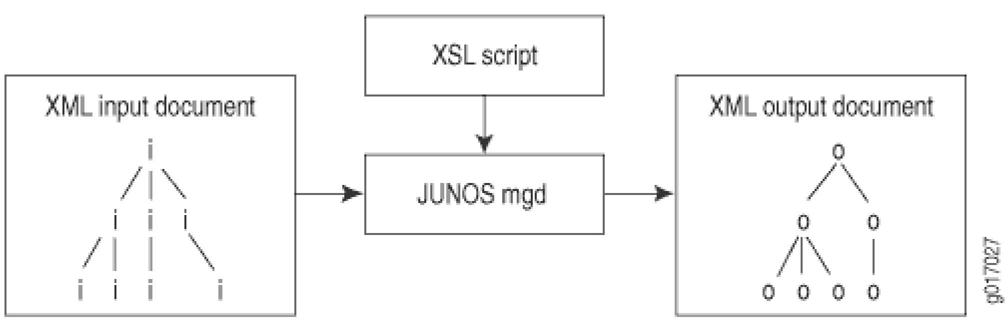
```
<xsl:template match="/">
```

Description

The `<xsl:template match="/">` template is an unnamed template in the **junos.xsl** file that allows you to use shortened *XPath* expressions in commit 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.

Junos OS provides XML-formatted input to a script. Commit script input consists of an XML representation of the post-inheritance candidate configuration file. When you execute a script, the Junos OS management process (*mgd*) generates an XML-formatted output document as the product of its evaluation of the input document, as shown in Figure 2 on page 72.

Figure 2: 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

xsl:text

IN THIS SECTION

Syntax

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

Description

Insert literal text in the output.

Usage Examples

See "Example: Require and Restrict Configuration Statements" on page 790, "Example: Impose a Minimum MTU Setting" on page 732, "Example: Limit the Number of E1 Interfaces" on page 742, "Example: Control IS-IS and MPLS Interfaces" on page 710, and "Example: Adding a Final then accept Term to a Firewall" on page 657.

xsl:value-of

IN THIS SECTION

Syntax

```
<xsl:value-of select="expression"/>
```

Description

Extract the value of an XML element and insert it into the output. The `select` attribute specifies the *XPath* expression that is evaluated. In the XPath 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 XPath expression specifying the node or attribute to evaluate.

Usage Examples

- "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674
- "Example: Configure Administrative Groups for LSPs" on page 684
- "Example: Control IS-IS and MPLS Interfaces" on page 710
- "Example: Impose a Minimum MTU Setting" on page 732
- "Example: Limit the Number of E1 Interfaces" on page 742

RELATED DOCUMENTATION

xsl:copy-of | 60

xsl:variable

IN THIS SECTION

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 that includes 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 (Optional) 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: Limit the Number of E1 Interfaces" on page 742, "Example: Limit the Number of ATM Virtual Circuits" on page 736, "Example: Configure Administrative Groups for LSPs" on page 684, and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

RELATED DOCUMENTATION

xsl:when

IN THIS SECTION

Syntax

```
<xsl:when test="boolean-expression">
    ...
</xsl:when>
```

Description

Within an `<xsl:choose>` instruction, specify a set of processing instructions that are executed when the expression specified in the `test` attribute evaluates to `TRUE`. 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 `<xsl:otherwise>` element, if there is one, is processed.

Attributes

test	Specifies a Boolean expression.

Usage Examples

- "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674
- "Example: Configure Dual Routing Engines" on page 696
- "Example: Prevent Import of the Full Routing Table" on page 781

RELATED DOCUMENTATION

XSLT Programming Instructions Overview | 34

xsl:choose | 57

xsl:for-each | 62

xsl:if | 63

xsl:otherwise | 66

xsl:with-param

IN THIS SECTION

Syntax

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

Description

Specify a parameter to pass into a template. This element can be used when applying templates with the `<xsl:apply-templates>` instruction or when calling templates with the `<xsl:call-template>` instruction.

Attributes

name Specifies the name of the parameter.

select (Optional) *XPath* expression specifying 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 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: Configure Dual Routing Engines" on page 696, "Example: Prevent Import of the Full Routing Table" on page 781, and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

RELATED DOCUMENTATION

XSLT Templates Overview | 25
xsl:apply-templates | 55
xsl:call-template | 56
xsl:param | 67
xsl:template | 70
xsl:variable | 77

3

PART

Automation Scripting Using SLAX

CHAPTER 6

SLAX Overview

IN THIS CHAPTER

SLAX Overview

IN THIS SECTION

Stylesheet Language Alternative syntaX (SLAX) is a language for writing Junos OS commit scripts, op scripts, event scripts, and SNMP 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.

Benefits of SLAX and SLAX Scripts

- SLAX scripts employ a clean, readable syntax that emphasizes XPath expressions and XML content creation.
- SLAX scripts provide fast configuration database access and XML processing for optimal performance.
- SLAX scripts can utilize the debugger and profiler tools in the libslax library to help parse, troubleshoot, and analyze the performance of the scripts to ensure the most efficient operation.

SLAX Syntax 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 enabling you to create readable, maintainable commit, op, event, and SNMP 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 the SLAX syntax are particularly strong for programmers who are not already accustomed to XSLT, because SLAX enables them to concentrate on the new programming topics introduced by XSLT, rather than concentrating on learning a new syntax. For example, SLAX enables you to:

- Write more readable scripts
- Reduce the clutter in your scripts
- Simplify namespace declarations
- Define named templates with a syntax resembling a function definition
- Invoke named templates with a syntax resembling a function call
- Use curly braces to show containment instead of closing tags
- 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 (=)
- Perform concatenation using the underscore (_) operator, as in PERL, version 6
- Write text strings using simple quotation marks (" ") instead of the `<xsl:text>` element

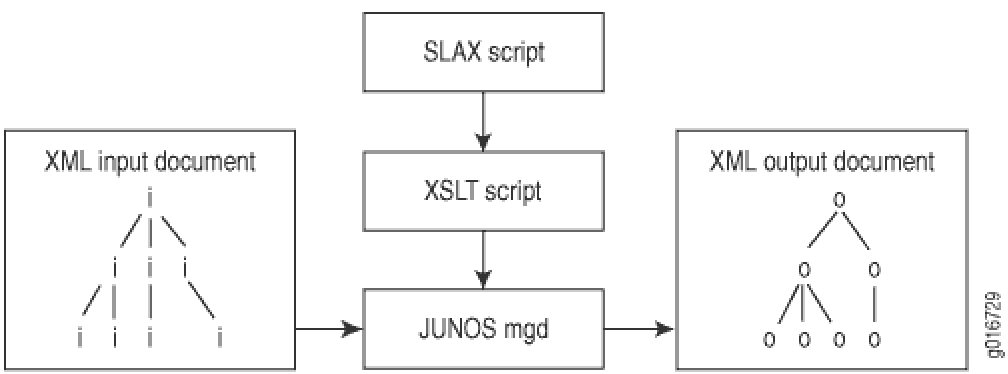
How SLAX Works

SLAX functions as a preprocessor for XSLT. 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—the *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 3 on page 85 shows the flow of SLAX script input and output.

Figure 3: SLAX Script Input and Output



SLAX Resources

Table 4 on page 86 outlines additional resources that you can use to learn SLAX and write SLAX scripts.

Table 4: SLAX Resources

Resource	URL
SLAX Manual	http://juniper.github.io/libslax/slax-manual.html
Junos Automation Reference for SLAX 1.0	https://www.juniper.net/documentation/en_US/day-one-books/archive/TW_Junos_Automation_Reference.pdf
Libslax—an open-source implementation of the SLAX language	https://github.com/Juniper/libslax https://github.com/Juniper/libslax/wiki

RELATED DOCUMENTATION

Convert Scripts Between SLAX and XSLT | 524

SLAX Syntax Rules Overview

IN THIS SECTION

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

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 1.2;

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 | 89
SLAX Overview | 83
SLAX Statements Overview | 114
SLAX Templates Overview | 97
SLAX Variables Overview | 110

SLAX Elements and Element Attributes Overview

IN THIS SECTION

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 | 9

SLAX Elements as Function Arguments

IN THIS SECTION

Starting with version 1.2 of the SLAX language, which is supported in Junos OS Release 14.2 and later releases, you can use SLAX elements directly as arguments for both functions and templates. Function arguments can either be a single element or a block of SLAX code, placed inside braces as shown in the following example:

```
var $a = my:function(<elt>,<max> 15);
```

```
var $b = my:test ({
    <min> 5;
    <max> 15;
    if ($step) {
        <step> $step;
    }
});
```

```
var $c = my:write(<content> {
    <document> "total.txt";
    <size> $file/size;
    if (node[@type == "full"]) {
        <full>;
    }
});
```

For templates, you still need to include the argument name, but the value can be inline. For example:

```
call my:valid($name = <name> $input, $object = {
    <min> $min;
    <max> $max;
});
```

The Main Template

Starting with version 1.2 of the SLAX language, which is supported in Junos OS Release 14.2 and later releases, the **main** template is introduced in SLAX. The `main` template enables you to process the input XML document and provide the top-level element for the output hierarchy. The `main` template is equivalent to using `match /` but slightly more convenient.

You can use the `main` statement to match the top of the input data hierarchy and create the top-level tag of the output hierarchy. You can use the statement in two forms: with or without the output tag. When you omit the output element, `main` is just followed by a block of statements within a set of braces, as shown in the following example:

```
main {
    <top> {
        <answer> 42;
    }
}
```

The `main` template can also be used with a top-level output element following the `main` token, as shown in the following example:

```
main <top> {
    <answer> 42;
}
```

Both of the preceding examples are equivalent to the following XSLT version:

```
<xsl:template match="/">
    <top>
        <answer>42</answer>
    </top>
</xsl:template>
```

RELATED DOCUMENTATION

SLAX Elements and Element Attributes Overview | 89

Understanding SLAX Default Namespaces

Version 1.2 of the SLAX language, which is supported in Junos OS Release 14.2 and later releases, introduces default namespaces for prefixes. Table 5 on page 94 lists the prefixes that have default namespaces installed with the libslax software distribution.

When a prefix is used without the corresponding `ns` statement in scope, SLAX refers to the set of default namespaces. If the prefix has a default namespace, that namespace is automatically mapped to the prefix.

Table 5: Prefix Set with libslax

Prefix	Source	Default URI
bit	libslax	xml.libslax.org/bit
curl	libslax	xml.libslax.org/curl
exsl	exslt	http://exslt.org/common
crypto	exslt	http://exslt.org/crypto
math	exslt	http://exslt.org/math
set	exslt	http://exslt.org/sets
func	exslt	http://exslt.org/functions
str	exslt	http://exslt.org/strings
date	exslt	http://exslt.org/dates-and-times
dyn	exslt	http://exslt.org/dynamic
saxon	libxslt	http://icl.com/saxon
os	libslax	http://xml.libslax.org/os
xutil	libslax	http://xml.libslax.org/xutil

When using the **slaxproc** tool with the `--format` or `--slax-to-xslt` command-line options, the namespace is properly displayed, as shown in the following example:

```
% cat /tmp/foo.slax
version 1.1;
```

```
match / {
    <top> {
        expr date:time();
    }
}
```

```
% slaxproc --format /tmp/foo.slax
version 1.1;
ns date extension = "http://exslt.org/dates-and-times";
match / {
    <top> date:time();
}
```

RELATED DOCUMENTATION

SLAX Syntax Rules Overview | 86

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() | 41
SLAX Elements and Element Attributes Overview | 89
SLAX Syntax Rules Overview | 86
XPath Overview | 22
xsl:text | 75
xsl:value-of | 76

SLAX Templates Overview

IN THIS SECTION

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.

NOTE: Version 1.2 of the SLAX language, which is supported in Junos OS Release 14.2 and later releases, supports SLAX elements as arguments to both templates and functions.

There are two types of templates, named and unnamed (or match), 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 and 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 `<div class="author">` 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. Alternatively, 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 104.

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 Functions Overview

Version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases, supports functions. When the complexity of a script increases or a code segment appears in multiple places, you can modularize the code and create functions. Functions accept arguments and run only when explicitly called.

NOTE: Version 1.2 of the SLAX language, which is supported in Junos OS Release 14.2 and later releases, supports SLAX elements as arguments to both templates and functions.

Functions have several advantages over templates, including the following:

- Arguments are passed by position rather than name.
- Return values can be objects as opposed to result tree fragments.
- Functions can be used in expressions.
- Functions can be resolved dynamically (using EXSLT `dyn:evaluate()`).

In SLAX, you define a function definition as a top-level statement in the script. The function definition consists of the `function` keyword, the function name, a set of arguments, and a braces-delimited block of code. The function name must be a qualified name. The argument list is a comma-separated list of parameter names, which are positionally assigned based on the function call. Trailing arguments can have default values. Alternatively, you can define function parameters inside the function block using the `param` statement. The syntax is:

```
function function-name (argument-list) {
    ...
```

```
    result return-value;
}
```

```
function function-name () {
    param param-name1;
    param param-name2;
    param param-name3 = default-value;
    ...
    result return-value;
}
```

The return value can be a scalar value, an XML element or XPath expression, or a set of instructions that emit the value to be returned.

If there are fewer arguments in the function invocation than in the definition, the default values are used for any trailing arguments. If there are more arguments in the function invocation than in the definition, the function call generates an error.

The following example defines the function `size`, which has three parameters: `width`, `height`, and `scale`. The default value for `scale` is 1. If the function call argument list does not include the `scale` argument, the calculation uses the default value of 1 for that argument. The function's return value is the product of the `width`, `height`, and `scale` variables enclosed in a `<size>` element.

In the main match template, the function call uses width and height data selected from each `graphic/dimension` element in the source XML file. The script evaluates the function, and the `copy-of` statement emits the return value to the result tree as the contents of the `<out>` element.

```
version 1.1;
ns my = "http://www.example.com/myfunctions";

function my:size ($width, $height, $scale = 1) {
    result <size> {
        expr $width * $height * $scale;
    }
}

match / {
    for-each (graphic/dimension) {
        <out> {
            copy-of my:size((width/.), (height/.));
        }
```

```
    }
}
```

The following function definition uses `param` statements to define the parameters instead of a comma-separated list. The behavior of the function is identical to that in the previous example.

```
version 1.1;
ns my = "http://www.example.com/myfunctions";

function my:size () {
    param $width;
    param $height;
    param $scale = 1;
    result <size> {
        expr $width * $height * $scale;
    }
}

match / {
    for-each (graphic/dimension) {
        <out> {
            copy-of my:size((width/.), (height/.));
        }
    }
}
```

RELATED DOCUMENTATION

function | 160
param | 187
result | 195

SLAX Parameters Overview

IN THIS SECTION

Parameters may be passed to named or unnamed templates or to functions. After declaring a parameter, you can reference it by prefixing the parameter name with the dollar sign ($).

Declaring Parameters

In SLAX scripts, 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 local or global. Local parameters must be declared at the beginning of a block, and their scope is limited to the block in which they are declared. A parameter whose value is set by Junos OS at script initialization must be defined as a global parameter. Global parameter declarations are placed just after the style sheet declarations. A script can assign a default value to the global parameter, which is used in the event that Junos OS does not give a value to the parameter.

```
version 1.2;
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";
```

In a template, you declare parameters either in a parameter list or by using the `param` statement in the template block. Optionally, you can declare default values for each template parameter. If a template is invoked without the parameter, the parameter uses the default value. 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;
   }
}
```

In the SLAX template, you prefix the parameter with the dollar sign ($) when you declare the parameter and when you access its value. In XSLT, you prefix the parameter name with the dollar sign when you access it but not when you declare it.

In a function, you declare parameters either in a parameter list or by using the `param` statement in the function block. Optionally, you can declare default values for trailing parameters. If you invoke a function without that trailing parameter, the parameter uses the default value. If you do not define a default value, the parameter value defaults to an empty string.

The following example defines a function named `size`, which has three parameters: `width`, `height`, and `scale`. The default value for `scale` is 1. If the function call's argument list does not include the `scale` argument, the calculation uses the default value of 1 for that argument. The return value for the function is the product of the `width`, `height`, and `scale` variables enclosed in a `<size>` element.

```
function my:size ($width, $height, $scale = 1) {
    result <size> {
        expr $width * $height * $scale;
    }
}
```

An alternative, but equivalent declaration, which uses the `param` statement, is:

```
function my:size () {
    param $width;
    param $height;
    param $scale = 1;
    result <size> {
        expr $width * $height * $scale;
    }
}
```

Passing Parameters to Templates

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 parameter uses the default value defined in the template for that parameter. If there is no default value for that parameter in the template, the parameter value 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 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. If the parameter was never declared, the script generates an error.

The following example calls the named template `print-host-name` and passes in the `message` parameter but does not include a value. If the script declares and initializes `message`, and the scope is visible to the block, the template uses the current value of `message`. If the script declares `message` but does not initialize the parameter, the value of `message` is an empty string. If the script does not declare `message`, the call 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>
```

Passing Parameters to Functions

SLAX supports functions starting in SLAX version 1.1. Although you can use the `param` statement to define function parameters, you cannot use the `with` statement to pass parameter values into the function from the calling environment. When you call a function, you pass arguments into the function in a comma-separated list. Function arguments are passed to the function by position rather than by name as in a template.

A function declaration can define default values for trailing arguments. If there are fewer arguments in the function invocation than in the definition, the default values are used for any trailing arguments. If there are more arguments in the function invocation than in the definition, the function call generates an error.

In the following match template, the function call uses width and height data selected from each `graphic/dimension` element in the source XML file. The script evaluates the function, and the `copy-of` statement emits the return value to the result tree as the contents of the `<out>` element. The function call includes arguments for `width` and `height`, but not for `scale`. The default value of 1 is used for `scale` within the function block.

```
version 1.2;
ns my = "http://www.example.com/myfunctions";

function my:size () {
    param $width;
    param $height;
    param $scale = 1;
    result <size> {
        expr $width * $height * $scale;
    }
}

match / {
    for-each (graphic/dimension) {
        <out> {
            copy-of my:size((width/.), (height/.));
        }
    }
}
```

RELATED DOCUMENTATION

SLAX Variables Overview

IN THIS SECTION

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.

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 script. The scope of a local variable is limited to the template or code block in which it is defined.

Version 1.0 of the SLAX language supports immutable variables, which are declared using the `var` statement. Version 1.1 of the SLAX language, which is supported starting with Junos OS Release 12.2, introduces mutable variables, which are declared using the `mvar` statement. Mutable and immutable variables are discussed in the following sections:

Immutable variables

In version 1.0 of the SLAX language, you declare variables using the `var` statement. Variables declared using the `var` statement are immutable. You can set the value of an immutable variable only when you declare it, after which point the value is fixed.

In the declaration, the variable name is prefixed with the dollar sign ($), which is unlike the XSLT declaration, where the dollar sign does not prefix the value of the `name` attribute of the `<xsl:variable>` element. Once you declare a variable, you can reference it within an XPath expression using the variable name prefixed with a dollar sign ($). 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`, which is then used to initialize the variable `message`:

```
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.')"/>
```

Variables declared using the `var` statement are immutable. As such, you can never change the value of the variable after it is initialized in the declaration. Although you cannot directly update the value of the 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
```

Mutable variables

Version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases, introduces mutable variables. Unlike variables declared using the `var` statement, the value of a mutable variable can be modified by a script. You can set the initial value of a mutable variable at the time you declare it or at any point in the script.

You declare mutable variables using the `mvar` statement. In the declaration, the variable name is prefixed with the dollar sign ($). Once you declare a mutable variable, you can reference it within an XPath expression using the variable name prefixed with a dollar sign ($). You initialize the variable by following the variable name with an equal sign (=) and an expression.

The following example declares and initializes the mutable variable `location`, which is then used to initialize the mutable variable `message`:

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

Mutable variables can be initialized or updated after they are declared. To initialize or update the value of a mutable variable, use the `set` statement. The following example declares the variable, `block`, and initializes it with the element `<block>`:

```
mvar $block;
set $block = <block> "start here";
```

For mutable variables that represent a node set, use the `append` statement to append a new node to the existing node set. The following example creates the mutable variable `$mylist`, which is initialized with one `<item>` element. For each grocery item in the `$list` variable, the script appends an `<item>` element to the `$mylist` node set with the name and size of the item.

```
version 1.1;

var $list := {
  <list> {
      <grocery> {
          <name> "milk";
          <type> "nonfat";
```

```
            <brand> "any";
            <size> "gallon";
        }
        <grocery> {
            <name> "orange juice";
            <type> "no pulp";
            <brand> "any";
            <size> "half gallon";
        }
        <drugstore>{
            <name> "aspirin";
            <brand> "any";
            <size> "50 tablets";
        }
    }
}

match / {

    mvar $mylist;
    set $mylist = <item> {
        <name> "coffee";
        <size> "1 lb";
    }
    for $item ($list/list/grocery) {
        append $mylist += <item> {
            <name> $item/name;
            <size> $item/size;
        }
    }
    <grocery-short-list> {
        copy-of $mylist;
    }
}
```

The output from the script is:

```
<?xml version="1.0"?>
<grocery-short-list>
  <item>
    <name>coffee</name>
    <size>1 lb</size>
```

```
  </item>
  <item>
    <name>milk</name>
    <size>gallon</size>
  </item>
  <item>
    <name>orange juice</name>
    <size>half gallon</size>
  </item>
</grocery-short-list>
```

RELATED DOCUMENTATION

SLAX Statements Overview

IN THIS SECTION

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

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` statements 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 97.

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 = "https://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. Supported versions include 1.0, 1.1, and 1.2.

```
version 1.2;
```

RELATED DOCUMENTATION

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 in SLAX version 1.0 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 120 summarizes the available operators and provides an example and an explanation of each.

Table 6: SLAX Operators

Name	Operator	Example / Explanation
Addition	+	`var $result = 1 + 1;` Return the sum of the operands. This example assigns a value of 2 to the `$result` variable.

Table 6: SLAX Operators *(Continued)*

Name	Operator	Example / Explanation
And	&&	`($byte-count > 500000) && ($byte-count < 1000000)` Evaluate two expressions and return one boolean result. If either of the two expressions evaluates to `false`, then the combined expression evaluates to `false`.

Assignment	=	`var $mtu = 1500;`
		`mvar $mtu2 = 48;`
		`set $mtu2 = 1500;`
		
		Assign a value to a variable or parameter or assign a namespace to a prefix. The example assigns a value of 1500 to both the `$mtu` variable and the `$mtu2` mutable variable. `$mtu2` was originally initialized with a value of 48.

Conditional	?:	`var $result = ($a < 10) ? $b : $c;` Provide conditional assignment based on the boolean value of the evaluated condition. If the conditional expression evaluates to `true`, the entire expression assumes the value of the operand to the left of the colon. If the conditional expression evaluates to `false`, the entire expression assumes the value of the operand to the right of the colon. This operator was introduced in version 1.1 of the SLAX language, which is supported starting with Junos OS Release 12.2. In the example, if the value stored in the variable `$a` is less than 10, `$result` is assigned the value stored in `$b`. Otherwise, `$result` is assigned the value stored in `$c`.
Division	div	`<output>$bit-count div 8;` Return the result of dividing the left operand by the right operand. If the remainder of the division is nonzero, the result is expressed in decimal floating-point notation. The example divides the `$bit-count` variable by eight, returning the byte count (requires that `$bit-count` has been initialized).

Table 6: SLAX Operators *(Continued)*

Name	Operato r	Example / Explanation
Equality	==	`$mtu == 1500` Return `true` if the values of the left and right operands are equal; otherwise, the expression returns `false`. In the example, if `$mtu` equals 1500, then the expression resolves to `true`; otherwise, it returns false (requires that `$mtu` has been initialized).
Greater than	>	`$hop-count > 0` Return `true` if the value of the left operand is greater than the value of the right operand; otherwise, the expression 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	>=	`$hop-count >= 1` Return `true` if the value of the left operand is either greater than or equal to the value of the right operand; otherwise, the expression returns `false`. In this example, if `$hop-count` is 1 or greater, the expression returns `true` (requires that `$hop-count` has been initialized).
Inequality	!=	`$mtu != 1500` Return `true` if the values of the left and right operands are not equal; otherwise, the expression returns `false`. In the example, if `$mtu` does not equal 1500, then the expression resolves to `true`; otherwise, the expression returns `false` (requires that `$mtu` has been initialized)

Table 6: SLAX Operators *(Continued)*

Name	Operator	Example / Explanation

Iteration	...	`for $i (1 ... 10) {`
		`  <player number=$i>;`
		`}`
		
		Iterate through a range of integer values with a start value equal to the left operand and an end value equal to the right operand. If the left operand is greater than the right, the numbers are generated in decreasing order. The operator translates into an XPath function that generates the sequence as a node set. This operator was introduced in version 1.1 of the SLAX language, which is supported starting with Junos OS Release 12.2.

Less than	<	`$hop-count < 15` Return `true` if the value of the left operand is less than the value of the right operand; otherwise, the expression returns `false`. In this example, if `$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` Return `true` if the value of the left operand is either less than or equal to the value of the right operand; otherwise, the expression returns `false`. In this example, if `$hop-count` is 14 or less, the expression returns `true` (requires that `$hop-count` has been initialized).
Modulo	mod	`<output> 10 mod 3;` Return the division remainder of two numbers. In this example, the expression outputs a value of 1.
Multiplication	*	`<output> 5 * 10;` Return the product of the operands. In this example, the expression outputs a value of 50.

Table 6: SLAX Operators *(Continued)*

Name	Operator	Example / Explanation

Node Set, append to	+=	`mvar $block = <block> "start here";`
		`append $block += <block> "next block";`
		
		Append a value to a node set contained in a mutable variable, which is defined with the `mvar` statement. This operator was introduced in version 1.1 of the SLAX language, which is supported starting with Junos OS Release 12.2.

Node Set Conversion	:=	`var $new-node-set := $rtf-variable;` Convert a result tree fragment into a node set. A result tree fragment contains an unparsed XML data structure. It is not possible to retrieve any of the embedded XML information from this data type. A script can convert the result tree fragment into a node set and then search the node set for the appropriate information and extract it. This operator is supported in Junos OS Release 9.2 and later releases.
Or	||	`($mtu-size != 1500) \|\| ($mtu-size > 2000)` Evaluate two expressions and return one boolean result. If either of the two expressions evaluates to true, then the combined expression evaluates to true.
Parentheses	( )	`var $result = ( $byte-count * 8 ) + 150;` Create complex expressions. Parentheses function the same way as in a mathematical expression, where the expression within the parentheses is evaluated first. Parentheses can be nested; the innermost set of parentheses is evaluated first, then the next set, and so on.
String concatenation	_ (under-score)	`var $combined-string = $host-name _ " is located at " _ $location;` Concatenate multiple strings (note that strings cannot be combined using the + operator in SLAX). In the example, if `$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
Subtraction	-	`var $result = 64 - 14;` Return the difference between the left operand and the right operand. This example assigns a value of 50 to the `$result` variable.

Unary Minus	-	`mvar $number = 5.`
		`set $number = - $number;`
		
		Negate the value of the operand, changing a positive value to a negative value or a negative value to a positive value. The example negates the value stored in `$number` and reassigns the new value of -5 to the variable.

Union	|	`var $all-interface-nodes = $fe-interface-nodes | $ge-interface-nodes;` Create 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.

RELATED DOCUMENTATION

SLAX Elements and Element Attributes Overview | 89
SLAX Overview | 83
SLAX Statements Overview | 114
SLAX Syntax Rules Overview | 86
SLAX Variables Overview | 110

CHAPTER 7

SLAX Statements

IN THIS CHAPTER

append

IN THIS SECTION

Syntax

```
append name += value;
```

Description

Append a value to the node set contained in a mutable variable. The variable must be defined using the `mvar` statement.

Attributes

name	Name of the mutable variable.
value	Value to append to the node set.

SLAX Example

The following snippet appends the `<item>` element and `<name>` and `<size>` child elements to the node set contained in the mutable variable `block`:

```
mvar $block;
set $block = <block> "item list";

for $item (list) {
    append $block += <item> {
        <name> $item/name;
        <size> $item/size;
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

apply-imports

IN THIS SECTION

Syntax

```
apply-imports;
```

Description

Apply a template rule from an external file or style sheet. By default, template rules in the main script have precedence over equivalent imported template rules. Use this statement to process the context node using the imported match template rule from the external source.

The `apply-imports` statement mimics the `<xsl:apply-imports>` element, allowing the script to invoke any imported templates.

SLAX Example

In the example, the main script imports the file **route-rules.slax**. The `apply-imports` statement invokes the imported template rule for `<route>` elements.

```
version 1.1;
import "route-rules.slax";

match route {
    <routes> {
        apply-imports;
    }
}
```

The imported file contains a template rule for `<route>` elements.

```
/* route-rules.slax */
version 1.1;

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

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

import | 165

apply-templates

IN THIS SECTION

Syntax

```
apply-templates expression;
```

Description

Apply one or more templates, according to the value of the node-set expression. If a node-set expression is not specified, the script recursively processes all child nodes of the current node. If a node-set expression is specified, the processor only applies templates to the child elements that match the node-set expression. The `template` statement dictates which elements are transformed according to which template. The templates that are applied are passed the parameters specified by the `with` statement within the `apply-templates` statement block.

Attributes

`expression` (Optional) 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 657 and "Example: Prevent Import of the Full Routing Table" on page 781.

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

attribute

IN THIS SECTION

Syntax

```
attribute attribute-name {
    attribute-value;
}
```

Description

Create an attribute with the given name. The attribute value is defined by a block of statements, which must be placed inside a set of braces.

Attributes

attribute-name	Name of the attribute, which can be an XPath expression or a string. Enclose string arguments in quotes.
attribute-value	A block of statements enclosed in curly braces that defines the attribute value.

SLAX Example

In the following example, the <book> element is output to the result tree with an attribute named format, which has the value "PDF":

```
<book> {
    attribute "format" {
        expr "PDF";
    }
}
```

In the following example, the value of the <name> node (rather than the literal string "name") is used to create an XML attribute with a value of "from-" concatenated with the contents of the address node. Node values are selected from the current context.

```
<source> {
    attribute name {
        expr "from-" _ address;
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

SLAX Elements and Element Attributes Overview | 89
attribute-set | 135
element | 146
use-attribute-sets | 210

attribute-set

IN THIS SECTION

Syntax

```
attribute-set attribute-set-name {;
    attribute attribute-name1 { attribute-value1; }
    attribute attribute-name2 { attribute-value2; }
    use-attribute-sets attribute-set-name2;
...
}
```

Description

Define a collection of attributes that can be used repeatedly. The `attribute-set` statement must be defined as a top-level statement in the script. The attribute set name is a string argument. The attribute set contents define the attributes to include in the collection. The contents can include individual `attribute` statements, which define attributes as a name and value pair, and they can include `use-attribute-sets` statements, which add the attributes from a previously defined attribute set to the current set.

To apply the attributes in an attribute set to a specific element, include the `use-attribute-sets` statement under that element and reference the attribute set name.

Attributes

attribute-set-name Name of the attribute set, which must be a string. To add the attribute set to an element, reference this name in the `use-attribute-sets` statement.

attribute-name Name of the individual attribute to add to the set.

attribute-value A block of statements enclosed in curly braces that defines the attribute value.

SLAX Example

The following example creates two attribute sets: `table-attributes` and `table-attributes-ext`. The `table-attributes-ext` set includes all of the attributes that are already defined in the `table-attributes` set through use of the `use-attributes-sets` statement. In the main script body, the `table-attributes-ext` attribute set is applied to the `<table>` element. The `<table>` element includes the four attributes: `order`, `cellpadding`, `cellspacing`, and `border`.

```
version 1.1;

var $cellpadding = "0";
var $cellspacing = "10";

attribute-set table-attributes {
    attribute "order" { expr "0"; }
    attribute "cellpadding" { expr $cellpadding; }
    attribute "cellspacing" { expr $cellspacing; }
}
attribute-set table-attributes-ext {
    use-attribute-sets table-attributes;
    attribute "border" { expr "0"; }
}

match / {
    ...
    <table> {
        use-attribute-sets table-attributes-ext;
    }
}
```

XSLT Equivalent

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:variable name="cellpadding" select="0"/>
```

```
    <xsl:variable name="cellspacing" select="10"/>
    <xsl:attribute-set name="table-attributes">
        <xsl:attribute name="order">
            <xsl:text>0</xsl:text>
        </xsl:attribute>
        <xsl:attribute name="cellpadding">
            <xsl:value-of select="$cellpadding"/>
        </xsl:attribute>
        <xsl:attribute name="cellspacing">
            <xsl:value-of select="$cellspacing"/>
        </xsl:attribute>
    </xsl:attribute-set>
    <xsl:attribute-set name="table-attributes-ext"
        use-attribute-sets="table-attributes">
        <xsl:attribute name="border">
            <xsl:text>0</xsl:text>
        </xsl:attribute>
    </xsl:attribute-set>
    <xsl:template match="/">
        <table use-attribute-sets="table-attributes-ext"/>
    </xsl:template>
</xsl:stylesheet>
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

SLAX Elements and Element Attributes Overview | 89
attribute | 133
element | 146
use-attribute-sets | 210

call

IN THIS SECTION

Syntax

```
call template-name (parameter-name = value) {
    /* code */
}
```

Description

Call a named template. You can pass parameters into the template by including a comma-separated list of parameters, with the parameter name and an optional equal sign (=) and value expression. If a value is not specified, the current value of the parameter is passed to the template.

You can declare additional parameters inside the code block using the `with` statement.

Attributes

template-name	Specifies the name of the template to call.

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: Require and Restrict Configuration Statements" on page 790, "Example: Impose a Minimum MTU Setting" on page 732, and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

copy-node

IN THIS SECTION

Syntax

```
copy-node;
```

```
copy-node {
    /* body */
}
```

Description

Copy the current node including namespace nodes to the result tree, but do not copy any attribute or child nodes. The optional body is a block of statements that emit additional nodes inside that copy.

SLAX Example

```
copy-node {
    <that> "one";
}
```

XSLT Equivalent

```
<xsl:copy>
    <that>
        <xsl:value-of select="one"/>
    </that>
</xsl:copy>
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

copy-of | 142

xsl:copy-of | 60

copy-of

IN THIS SECTION

Syntax

```
copy-of expression;
```

Description

Copy the specified node including namespace nodes, child nodes, and attributes of that node. The argument is an XPath expression that specifies which nodes to copy.

Attributes

expression	XPath expression that specifies which nodes to copy.

SLAX Example

```
copy-of configuration/protocols/bgp;
```

XSLT Equivalent

```
<xsl:copy-of select="configuration/protocols/bgp"/>
```

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

copy-node | 140
xsl:copy-of | 60

decimal-format

IN THIS SECTION

Syntax

```
decimal-format format-name {
    decimal-separator character;
    digit character ;
    grouping-separator character;
    infinity string;
    minus-sign character;
    nan string;
    pattern-separator character;
    percent character;
    per-mille character;
```

```
    zero-digit character;
}
```

Description

Define formatting parameters for use by the `format-number()` XPath function. The `decimal-format` statement must be defined as a top-level statement in the script.

Attributes

decimal-format ***format-name***	Decimal-format identifier, which is passed as the third argument to the `format-number()` XPath function.
decimal-separator ***character***	Character used as the decimal sign. The default is the period (.).
digit ***character***	Character used to represent a digit in a pattern. The default is the number sign (#).
grouping-separator ***character***	Character used as the digit group separator or the thousands separator. The default is the comma (,).
infinity ***string***	String used to represent infinity. The default is "Infinity".
minus-sign ***character***	Character used as the minus sign. The default is the hyphen (-).
nan ***string***	String used to represent NaN. The default is "NaN".
pattern-separator ***character***	Character used to separate patterns. The first pattern is used for positive numbers, and the second pattern is used for negative numbers. The default is the semicolon (;).
percent ***character***	Character used as the percent sign. The default is the percent character (%).
per-mille ***character***	Character used as a per mille sign. The default is the Unicode per mille sign (\x2030 or ‰).
zero-digit ***character***	Character used as zero. The default is the number zero (0).

SLAX Example

The following code snippet lists the defaults for the decimal-format parameters, and uses the defined decimal format in the `format-number` XPath function:

```
version 1.1;

decimal-format default-format {
    decimal-separator "." ;
    digit "#" ;
    grouping-separator "," ;
    infinity "Infinity" ;
    minus-sign "-" ;
    nan "NaN";
    pattern-separator ";" ;
    percent "%";
    per-mille "\x2030";
    zero-digit "0" ;
}

match / {
...
    var $number = -14560302.5;
     expr format-number($number, "###,###.00", "default-format");
}

/* output is -14,560,302.50 */
```

XSLT Equivalent

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:decimal-format name="default-format" decimal-separator="." digit="#"
      grouping-separator="," infinity="Infinity" minus-sign="-" NaN="NaN"
      pattern-separator=";" percent="%" per-mille="\x2030" zero-digit="0"/>

  <xsl:template match="/">
    <xsl:variable name="number" select="-14560302.5"/>
    <xsl:value-of select="format-number($number, '###,###.00',
```

```
        'default-format')"/>
    </xsl:template>
</xsl:stylesheet>
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

output-method | 183

element

IN THIS SECTION

Syntax

```
element name {
    /* element contents */
}
```

Description

Create an element node with the given name. The element name can be an XPath expression or a string. The element contents must be placed inside curly braces.

Attributes

name Name of the element, which can be an XPath expression or a string. Enclose string arguments in quotes.

SLAX Example

The following sample code uses the value of the `name` node (rather than the literal string "name") to create an XML element, whose contents are an empty element with a name of "`from-`" concatenated with the value of the address node. Node values are selected from the current context.

```
for-each (list/item) {
    element name {
        element "from-" _ address;
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

SLAX Elements and Element Attributes Overview | 89

else

IN THIS SECTION

Syntax

```
if (expression) {
        /* code */
}
else {
    /* code */
}
```

Description

Include a default set of instructions that are processed if the preceding `if` and `else if` statements evaluate to `FALSE`.

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: Configure Dual Routing Engines" on page 696 and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

SLAX Statements Overview | 114
else if | 150
for-each | 158
if | 163

else if

IN THIS SECTION

Syntax

```
if (expression) {
        /* code */
}
else if (expression) {
        /* code */
}
```

Description

Include instructions that are processed if the expression defined in the preceding `if` statement evaluates to `FALSE` and the expression defined in the `else if` statement evaluates to `TRUE`. Multiple `else if` statements can be included, but the processor only executes the instructions contained in the first `else if` statement whose expression evaluates to `TRUE`. All subsequent `else if` statements are ignored.

SLAX Example

```
var $description2 = {
   if (description) {
      expr description;
   }
   else if (../description) {
```

```
        expr ../description;
    }
    else {
        expr "no description found";
    }
}
```

XSLT Equivalent

```
<xsl:variable name="description2">
    <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:otherwise>unknown</xsl:otherwise>
    </xsl:choose>
</xsl:variable>
```

Usage Examples

See "Example: Configure Dual Routing Engines" on page 696 and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

expr

IN THIS SECTION

Syntax

```
expr expression;
```

Description

Generate the string value of an XPath expression and add it to the result tree. The XPath expression might consist of a function call, a location path, a literal number, or a string. SLAX-specific operators are permitted. This statement cannot be used at the top-level of a script. It can only appear within a code block. By default, characters such as "<", ">", and "&" are escaped into proper XML as "<", ">", and "&", respectively.

The `expr` statement is most commonly used to invoke functions that return no results, for conditional variable assignment, and to return text content from a template.

Attributes

expression	XPath expression to evaluate. The resulting string is added to the result tree.

SLAX Example

```
expr "Test: ";
expr substring-before(name, ".");
expr status;
expr jcs:output("Test");
```

XSLT Equivalent

```
<xsl:text>Test: </xsl:text>
<xsl:value-of select="substring-before(name, '.')"/>
<xsl:value-of select="status"/>
<xsl:value-of select="jcs:output('Test')"/>
```

Usage Examples

- "Example: Adding a Final then accept Term to a Firewall" on page 657
- "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

XPath Expressions Overview for SLAX | 95

message | 172

terminate | 205

trace | 206

uexpr | 208

fallback

IN THIS SECTION

Syntax

```
fallback {
   /* body to execute if extension function
      or element is unavailable */
}
```

Description

Specify statements to use when an extension function or element is not available in the current implementation. The `fallback` statement is enclosed within another instruction element to indicate what fallback code should be run if the script processor does not recognize the enclosing instruction element. The script executes the body of the fallback statement to handle this error condition.

A script might utilize this statement when it is run in environments that support different extension elements.

SLAX Example

The following example op script declares the namespace binding `test` with a URI of "test". The code attempts to reference the nonexistent extension element `<test:fake>`, which is not supported, and the code instead executes the fallback instructions.

```
version 1.1;
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 test extension = "test";


match / {
    <op-script-results> {
        /* Fake extension element */
        <test:fake> {
            expr slax:output( "<test:fake> exists!" );
            fallback {
                expr slax:output( "<test:fake> does not exist." );
            }
        }
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

for

IN THIS SECTION

Syntax

```
for name (expression) {
    /* code */
}
```

```
for name (min ... max) {
    /* code */
}
```

Description

Iterate through an integer set or a node set without changing the context, and execute a block of statements using each member of the integer or node set as the value of the given variable.

If the argument is an XPath expression, the variable is assigned each member of the node set selected by the expression in sequence. If the argument is an integer set, the iteration operator (...) generates a sequence of nodes with the value of each integer between the left and right operands. If the left operand is greater than the right operand, the numbers are generated in decreasing order. The variable takes on the value of each integer in sequence. For each iteration, the contents are then evaluated, processed according to the instructions contained in the `for` code block.

Attributes

expression XPath expression that selects the nodes to be processed.

max Integer or variable that defines the end value of the integer sequence. If the end value is less than the start value, the numbers are generated in decreasing order.

min Integer or variable that defines the starting value of the integer sequence. If the start value is greater than the end value, the numbers are generated in decreasing order.

name Identifier of the `for` loop variable, which takes on the values of each member of the integer or node set. This variable can be referenced within the `for` loop code block.

SLAX Example

In the following example, the `for` loop iterates over the `interfaces` node. The XPath expression selects each `name` node that is a child of the `interface` node and that has a value beginning with the 'ge-'

designator. The selection is assigned to the `$name` variable, which is used within that iteration of the `for` loop code block. The `for` loop outputs a `<name>` element for each selection. The content of each `<name>` element is the interface name currently stored in the `$name` variable for that iteration. The end result is a list of all Gigabit Ethernet interfaces on the device.

```
for $name (interfaces/interface[starts-with(name, 'ge-')]) {
    <name> {
        expr $name;
    }
}
```

In the following example, the `for` loop iterates over the integers 1 through 3, and the variable `$int` assumes each integer value. For each iteration, the code block generates an `<item>` element, which contains the attribute `item-number` with a value equal to the current integer value of `$int`.

```
for $int (1 ... 3) {
    <item> {
        attribute "item-number" {
            expr $int;
        }
    }
}

/* Output: <item item-number="1"/><item item-number="2"/><item item-number="3"/> */
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

SLAX Statements Overview | 114
XPath Overview | 22
for-each | 158

for-each

IN THIS SECTION

Syntax

```
for-each (expression) {
    /* code */
}
```

```
/* Syntax added in version 1.1 of the SLAX language.*/
for-each (min ... max) {
    /* code */
}
```

Description

Include a looping mechanism that repeats script processing for each XML element in the specified node set or each value in the integer set.

If the argument is an *XPath* expression, the element nodes are selected by the value of the XPath expression. If the argument is an integer set, the iteration operator (...) generates a sequence of nodes with the value of each integer between the left and right operands. If the left operand is greater than the right operand, the numbers are generated in decreasing order. For each iteration, the contents are then evaluated, processed according to the instructions contained in the `for-each` code block.

Attributes

`for-each` *`expression`* XPath expression that selects the nodes to be processed.

`max` Integer or variable that defines the end value of the integer sequence. If the end value is less than the start value, the numbers are generated in decreasing order.

`min` Integer or variable that defines the starting value of the integer sequence. If the start value is greater than the end value, the numbers are generated in decreasing order.

SLAX Example

The following code iterates over each `chassis-sub-module` element that has a part-number child element equal to 750-000610. For each match, the script outputs a `<message>` element with the name of the module and the name and description of the submodule.

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

The following code iterates over the integers 1 through 3. For each iteration, the code block generates an `<item>` element, which contains the attribute `item-number` with a value equal to the current integer value of the set.

```
for-each (1 ... 3) {
    <item> {
        attribute "item-number" {
            expr .;
        }
    }
}

/* Output: <item item-number="1"/><item item-number="2"/><item item-number="3"/> */
```

Usage Examples

- "Example: Adding T1 Interfaces to a RIP Group" on page 663
- "Example: Configure Administrative Groups for LSPs" on page 684

- "Example: Configure Dual Routing Engines" on page 696
- "Example: Impose a Minimum MTU Setting" on page 732
- "Example: Limit the Number of E1 Interfaces" on page 742
- "Example: Require and Restrict Configuration Statements" on page 790

Release Information

Statement introduced in version 1.0 of the SLAX language.

Support for iteration operator (...) added in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

SLAX Statements Overview | 114

XPath Overview | 22

for | 155

xsl:for-each | 62

function

IN THIS SECTION

Syntax

```
function function-name (argument-list) {
    ...
    result return-value;
}
```

```
function function-name () {
    param param-name1;
    param param-name2;
    param param-name3 = default-value;
    ...
    result return-value;
}
```

Description

Define an extension function that can be used in XPath expressions. The `function` statement must be defined as a top-level statement in the script using a qualified name for the function identifier. The argument list is a comma-separated list of parameter names, which are positionally assigned based on the function call. Trailing arguments can have default values. Alternatively, you can define function parameters inside the function block using the `param` statement. The function body is a set of statements, which should include a `result` statement that defines the return value for the function.

If there are fewer arguments in the function invocation than in the definition, the default values are used for any trailing arguments. If there are more arguments in the function invocation than in the definition, the function call generates an error.

Attributes

function-name Specifies the name of the function as a qualified name.

argument-list Comma-separated list of parameter names, which are positionally assigned based on the function call. Trailing arguments can have default values.

return-value XML element or XPath expression, scalar value, or a set of instructions providing the return value of the function.

SLAX Example

The following example defines the function `size`, which has three parameters: `width`, `height`, and `scale`. The default value for `scale` is 1. If the function call argument list does not include the `scale` argument, the calculation uses the default value of 1 for that argument. The function's return value is the product of the `width`, `height`, and `scale` variables enclosed in a `<size>` element.

In the main match template, the function call uses width and height data selected from each `graphic/dimension` element in the source XML file. The script evaluates the function, and the `copy-of` statement emits the return value to the result tree as the contents of the `<out>` element.

```
version 1.1;
ns my = "http://www.example.com/myfunctions";

function my:size ($width, $height, $scale = 1) {
    result <size> {
        expr $width * $height * $scale;
    }
}

match / {
    for-each (graphic/dimension) {
        <out> {
            copy-of my:size((width/.), (height/.));
        }
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

SLAX Functions Overview | 101
param | 187
result | 195

if

IN THIS SECTION

Syntax

```
if (expression) {
    /* code */
}
else if (expression) {
    /* code */
}
else {
    /* code */
}
```

Description

Include a conditional construct that causes instructions to be processed if the Boolean expression evaluates to TRUE.

Optionally, you can include multiple `else if` statements following an `if` statement to perform additional conditional tests if the expression in the `if` statement evaluates to `FALSE`. Multiple `else if` statements can be included, but the processor only executes the instructions contained in the first `else if` statement whose expression evaluates to `TRUE`; all subsequent `else if` statements are ignored. The optional `else` statement includes a default set of instructions that are processed if the expressions defined in all associated `if` and `else if` statements evaluate to `FALSE`.

Attributes

expression	Specifies the expression to evaluate.

SLAX Example

```
var $description2 = {
   if (description) {
      expr description;
   }
   else if (../description) {
      expr ../description;
   }
   else {
      expr "no description found";
   }
}
```

XSLT Equivalent

```
<xsl:variable name="description2">
   <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:otherwise>unknown</xsl:otherwise>
   </xsl:choose>
</xsl:variable>
```

Usage Examples

See "Example: Configure Dual Routing Engines" on page 696, "Example: Prevent Import of the Full Routing Table" on page 781, and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

import

IN THIS SECTION

Syntax

```
import href;
```

Description

Import rules from an external file or style sheet, which provide access to all the declarations and templates within the imported item. Any `import` statements must be the first elements in the script or style sheet. The path can be any URI. The path **../import/junos.xsl** is standard for all commit scripts, op scripts, and event scripts.

Imported rules are overwritten by any subsequent matching rules within the importing script. If more than one file or style sheet is imported, the items imported last override each previous import where the rules match.

Attributes

href Specifies the location of the imported file or style sheet.

SLAX Example

In the example, the main script imports the file **route-rules.slax**, which contains a template rule for <route> elements.

```
version 1.1;
import "route-rules.slax";

match route {
    <routes> {
        apply-imports;
    }
}
```

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

Understanding Named Templates in Junos OS Automation Scripts | 427
Global Parameters and Variables in Junos OS Automation Scripts | 327
Required Boilerplate for Commit Scripts | 542
Required Boilerplate for Event Scripts | 1084
Required Boilerplate for Op Scripts | 822
apply-imports | 129

key

IN THIS SECTION

Syntax

```
key name {
    match pattern;
    value expression;
}
```

Description

Define a key for use with the `key()` XPath function. Keys are an alternative to IDs and are used to index the nodes within an XML document. The key must be defined as a top-level statement in the script. A `key` definition consists of the key identifier, the nodes to index, and the value that is paired with the key name to reference the matching nodes. The `key()` function is then used to locate the appropriate nodes.

The `key()` function works with the XML document of the current node and uses the specified `key` definition to retrieve nodes that are referenced by a particular name and value. The function arguments are the key name and the desired key's value. The return value is a node set that includes all nodes referenced by that key name and value. If the desired key value is provided as a node set, rather than a string, the returned node set is a union of all the referenced nodes for the key values expressed by the nodes within the node set.

For example, if you define the key:

```
key func {
    match prototype;
```

```
    value @name;
}
```

the following code would select <prototype> elements that have a name attribute with a value of "trace", and then output the value of the child element <return-type>:

```
for-each ( key("func", "trace") ) {
    <out> return-type/.;
}
```

Attributes

key ***name***	Key identifier, which uniquely identifies the key within the script and is passed as the first argument to the key() function.
match ***pattern***	XPath expression that selects the set of nodes to index.
value ***expression***	XPath expression that defines the value of the key.

SLAX Example

The following op script creates two key definitions, protocol and next-hop, which are used to retrieve and display all static routes and all routes with a next hop of ge-0/0/0.0 on a device. The script invokes the Junos XML API get-route-information command to obtain the route information for the device. The for-each($results) statement changes the current node to the $results XML document. The subsequent for-each loops use the keys to retrieve all nodes that are indexed according to the key names and values.

The for-each(key("protocol", "Static") statement uses the protocol key definition, which matches on route-table/rt elements, to retrieve the desired nodes. The rt-entry/protocol-name key value matches the <protocol-name> child elements that have the value "Static". The code block executes using <rt> as the context node. For each match, the script outputs the value of the <rt-destination> element.

The for-each(key("next-hop", "ge-0/0/0.0") statement uses the "next-hop" key definition, which matches on route-table/rt elements, to retrieve the desired nodes. The rt-entry/nh/via key value matches the <via> child elements that have the value "ge-0/0/0.0". The code block executes using <rt> as the context node. For each match, the script outputs the value of the <rt-destination> element.

```
version 1.1;

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";

key protocol {
  match route-table/rt;
  value   rt-entry/protocol-name;
}
key next-hop {
  match route-table/rt;
  value   rt-entry/nh/via;
}

match / {
    <op-script-results> {
        var $results = jcs:invoke("get-route-information");

        for-each( $results ) {
           /* Display all static routes */
           <output> "Static routes: ";
           for-each( key( "protocol", "Static" ) ) {
              <output> rt-destination;
           }
           /* Display all routes with next-hop of ge-0/0/0.0 */
           <output> "Next-hop ge-0/0/0.0: ";
           for-each( key( "next-hop", "ge-0/0/0.0" ) ) {
               <output> rt-destination;
           }
        }
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

XPath Overview | 22
XPath Expressions Overview for SLAX | 95

match

IN THIS SECTION

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 root element of the document.

Attributes

expression *XPath* expression specifying the nodes to which to apply the template.

SLAX Example

```
match host-name {
    <hello> .;
}
```

XSLT Equivalent

```
<xsl:template match="host-name">
    <hello>
        <xsl:value-of select="."/>
    </hello>
</xsl:template>
```

Usage Examples

"Example: Adding a Final then accept Term to a Firewall" on page 657, "Example: Configure Dual Routing Engines" on page 696, and "Example: Prevent Import of the Full Routing Table" on page 781.

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

apply-templates | 131
call | 138
mode | 173
priority | 191
template | 203
with | 216

message

IN THIS SECTION

Syntax

```
message expression;
```

```
message {
  /* body */
}
```

Description

Generate an error message that is immediately displayed to the user, typically on the standard error file descriptor. This is different from most script output, which is displayed only after the script generates the final result tree.

Junos OS op scripts, event scripts, and commit scripts prepend "error:" to the displayed message when generating text output. When generating XML output, the scripts place the output inside a `<message>` element, which is enclosed in an `<xmn:error>` element.

If the `message` statement is used in a commit script, the script will generate two errors and terminate the commit process. If the `message` statement is used in an event script, the script writes the message to the output file, if one is configured.

Attributes

`message` *`expression`*	XPath expression or string emitted as output.

SLAX Example

```
if (not(valid)) {
    message "The " _ name() _ " node is not valid";
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

mode

IN THIS SECTION

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. If more than one template matches a node in the specified mode, the priority 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 `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 657.

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

mvar

IN THIS SECTION

Syntax

```
mvar $name[=initial-value];
```

Description

Declare a mutable variable in a SLAX script. You can initialize a mutable variable when you declare it by following the variable name with an equal sign (=) and a value.

Mutable variables differ from variables declared using the `var` statement in that you can change the value of a mutable variable after it is declared. To initialize or set the value of a mutable variable after you declare it, use the `set` statement. To append a value to the node set contained in a mutable variable, use the `append` statement.

NOTE: Mutable variables use non-standard SLAX specific extension elements, which can affect the portability of a script.

Attributes

name	Mutable variable identifier. After declaration, you can reference the variable within expressions by using the identifier prefixed with the dollar sign ($) character.
initial-value	Initial value assigned to the mutable variable.

SLAX Example

The following example creates the mutable variable `block`, and initializes it. The `set` statement assigns a new value to the `block` variable, overwriting the initial value set in the declaration. In the `for` loop, the code iterates over each item in the specified list and appends an `<item>` element with two child elements, `<name>` and `<size>`, to the node set stored in the `block` variable.

```
mvar $block= <block> "start here";
set $block = <block> "item list";

for $item (list) {
    append $block += <item> {
        <name> $item/name;
        <size> $item/size;
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

number

IN THIS SECTION

Syntax

```
number expression {
    format numbering-style;
    grouping-separator character;
    grouping-size number;
}


number {
    count nodes;
    format numbering-style;
    from nodes;
    grouping-separator character;
    grouping-size number;
    level "single" | "multiple" | "any";
}
```

Description

Generate a formatted number string, which is output to the result tree. When used with an argument, the statement formats the number given by that XPath expression. When used without an argument, the statement uses the `count`, `from`, and `level` options to generate the number based on the position of one or more nodes within the current XML document. In both cases, optional statements specify the formatting for that number. If needed, you can also redirect the formatted number string to a variable or output method instead of the result tree.

Attributes

`number` ***`expression`***
XPath expression providing the number to format.

`count` ***`nodes`***
XPath expression specifying which nodes should be counted. If `count` is omitted, it defaults to nodes with the same name as the current node.

`format` ***`numbering-style`***
A string, variable, or XPath expression that defines the number formatting.

The `format` option can include the following:

- start string–Any non-alphanumeric characters that precede the first number token in the format string. The start string is prepended to the formatted number string.

- number token–One or more number tokens that indicate what numbering format to use for the included numbers. The formatted number string only includes more than one number if the `level` option is set to “multiple”. Table 7 on page 179 lists format values and corresponding styles. The default value is “1”, which uses a decimal format style. When using decimal format, you can specify the minimum length of the formatted number string by preceding the “1” with one or more zeros.
- token separator–Non-alphanumeric characters that separate number tokens in the format string. These characters are included in the formatted number string between the computed numbers.
- end string–Any non-alphanumeric characters that follow the last number token in the format string. The end string is appended to the formatted number string.

Table 7: Numbering Styles for SLAX Statement number, format Option

Format Value	Style	Example
1	Decimal format	1 2 3 ...10 11 ...
01	Decimal format with a minimum output string length of 2	01 02 03 ... 10 11 ...
001	Decimal format with a minimum output string length of 3	001 002 003 ... 010 011 012 ... 100, 101
a	Lowercase alphabetic numbering	a b c ... z ... aa ab ... az ... ba bb ...
A	Uppercase alphabetic numbering	A B C ... Z ... AA AB ... BA BB ...
i	Lowercase Roman numbering	i ii iii iv v ...
I	Uppercase Roman numbering	I II III IV V ...

`from` ***`nodes`*** XPath expression specifying from which element to start the count. When `level` is set to `single` or `multiple`, this option constrains the counting to only node descendants of the nearest ancestor that matches the expression. When `level` is set to `any`, this option constrains the counting to only nodes that follow the nearest ancestor or preceding node of the current node that matches the expression.

grouping-separator ***character*** — Character used to delimit groups of digits for numbers expressed in decimal format. For example, decimal notation uses a comma as the delimiter between digit groupings.

grouping-size ***number*** — Defines the number of digits in a group for numbers expressed in decimal format. Setting this option causes the formatted number to be split into multiple groups according to the grouping size, with the grouping separator delimiting the groups. For example, decimal notation often uses a grouping size of 3.

level — Specifies what type of counting to perform. Accepted values are `single`, `multiple`, and `any`. The default is `single`. Specifying `single` starts the counting from the first ancestor node, specifying `multiple` starts the counting from any ancestor node, and specifying `any` starts the counting from any node.

- `single`–Perform only one count. The current node, if it matches the count expression, or the nearest ancestor that matches the count expression, is counted. The position of the node in document order, relative to its siblings that also match the count parameter, is used as the number to be formatted.

- `multiple`–Separately count all nodes that match the count expression and are either the current node or an ancestor of the current node. The position of each node in document order, relative to its siblings that also match the count parameter, is used as one of the numbers to be formatted.

- `any`–Perform only one count. The current node, if it matches the count pattern, or its nearest ancestor or preceding node that matches the count pattern, is counted. The position of the node in document order, relative to all other matching nodes that are ancestors or precede the node, is used as the number to be formatted.

> **NOTE**: Currently libxslt (1.1.26) does not support the “language” and “letter-value” options for the `<xsl:number>` element. While SLAX provides a means of encoding these XSLT constructs, they cannot be used in Junos OS.

SLAX Example

The following sample code iterates from 1 through 5. For each integer, the `number` statement outputs the equivalent uppercase Roman numeral value.

```
for $i (1 ... 5) {
  number $i {
    format "I ";
  }
}
```

```
I II III IV V
```

The following sample code provides the string “1234567890” to the `number` statement, which formats the output in decimal format with a group size of 3 and a comma as a group delimiter.

```
number "1234567890" {
    grouping-size 3;
    grouping-separator ",";
    format "1";
}
```

```
1,234,567,890
```

The following sample code counts all the `name` elements in the configuration hierarchy stored in the variable `$data`. The `count` option combined with the `level "multiple"` option tracks the count for any `name` elements under the `interface`, `unit`, and `address` elements.

The `format` option (1.A.a) includes a start string, which is an open parenthesis, and an end string, which is a close parenthesis and a space character. The number tokens are “1”, “A”, and “a”, which define the formatting of the numbers as decimal format, uppercase alphabetic numbering, and lowercase alphabetic numbering, respectively. The token separator is a period, which is also included in the output.

```
var $data := {
  <interfaces> {
      <interface> {
          <name> "ge-0/0/0";
          <unit> {
```

```
                <name> "0";
            }
            <unit> {
                <name> "1";
            }
       }
        <interface> {
            <name> "ge-0/1/0";
            <unit> {
                <name> "10";
                <family> {
                    <inet>;
                }
            }
        }
        <interface> {
            <name> "ge-2/0/2";
            <unit> {
                <name> "0";
                <family> {
                    <inet> {
                        <address> {
                            <name> "10.1.1.1/24";
                        }
                    }
                }
            }
        }
    }
}

for-each ($data//name) {
    number {
      level "multiple";
      count interface|unit|address;
      format "(1.A.a) ";
    }
    expr . _ "\n";
}
```

For the generated numbers displayed in the result tree, the decimal number in parentheses is associated with a particular interface. For each interface, the uppercase letter is associated with each logical unit name, and any lowercase letter is associated with the address `name` element for that logical unit, which is the IP address.

```
(1) ge-0/0/0
(1.A) 0
(1.B) 1
(2) ge-0/1/0
(2.A) 10
(3) ge-2/0/2
(3.A) 0
(3.A.a) 10.1.1.1/24
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

decimal-format | 143
output-method | 183

output-method

IN THIS SECTION

Syntax

```
output-method output-format {
    cdata-section-elements name-list;
    doctype-public string;
    doctype-system string;
    encoding string;
    indent "yes" | "no";
    media-type string;
    omit-xml-declaration "yes" | "no";
    standalone "yes" | "no";
    version string;
}
```

Description

Define the style used for result tree output. The `output-method` statement must be defined as a top-level statement in the script. Output formats include HTML, text, or XML. The default is XML, unless the first child element of the root node is `<html>` and there are no preceding text nodes, in which case the default output format is HTML.

Attributes

output-format Specify the format of the output. Acceptable values are “html”, “text”, “xml”, or a qualified name. The default is XML, unless the first child element of the root node is `<html>` and there are no preceding text nodes, in which case the default output format is HTML. Specifying a format of XML adds the XML declaration (`<?xml ?>`) to the result tree file.

cdata-section-elements *name-list* Specify a space-delimited list of the names of output elements whose text contents should be output to the result tree using CDATA sections. A CDATA section starts with "<![CDATA[" and ends with "]]>", and the contents of the section are interpreted by an XML parser as character data only, rather than markup.

doctype-public *string* Add the DOCTYPE declaration to the result tree, and specify the value of the `PUBLIC` attribute, which tells the parser where to locate the Document Type Definition (DTD) file.

doctype-system *string* Add the DOCTYPE declaration to the result tree, and specify the value of the `SYSTEM` attribute, which tells the parser where to locate the DTD file on the system.

encoding ***string***
: Explicitly add the pseudo-attribute `encoding` to the XML declaration in the output, and specify the character encoding used to encode the document, for example UTF-8, UTF-16, or ISO-8859-1.

indent "yes" | "no"
: Specify whether to indent the result tree output according to the hierarchical structure. Acceptable values are “yes” and “no”.

- **Default:** No indentation

media-type ***string***
: Define the MIME content type of the output.

- **Default:** text/xml

omit-xml-declaration "yes" | "no"
: Specify whether to include or omit the XML declaration (`<?xml ?>`) in the output.

- **Default:** no

standalone "yes" | "no"
: Explicitly add the pseudo-attribute `standalone` with the given string value to the XML declaration (`<?xml ?>`) in the output . Acceptable values are “yes” and “no”. The `standalone` attribute is only relevant if the document uses a DTD. If the `standalone` option is not included in the `output-method` statement, there is no explicit declaration in the result tree, which is identical to `standalone="no"`.

version ***string***
: For HTML and XML formats, set the W3C version for the output format. The pseudo-attribute `version` is included in the XML declaration (`<?xml ?>`) with the given version number.

SLAX Example

The following example uses the output method XML, which creates an XML declaration in the result tree output and adds the pseudo-attributes `version`, `encoding`, and `standalone` to the declaration. The DOCTYPE declaration has the root element `<html>` and provides values for both the `PUBLIC` and the `SYSTEM` attributes.

```
version 1.1;

output-method xml {
    doctype-public "-//W3C//DTD XHTML 1.0 Transitional//EN";
    doctype-system "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd";
    encoding "utf-8";
    indent "yes";
    omit-xml-declaration "no";
    standalone "no";
```

```
    version "1.0";
}

match / {
    <html> {
        <script type="text/javascript" src="/assets/js/api.js">;
     /* ... */
    }
}
```

The script produces the following output:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <script type="text/javascript" src="/assets/js/api.js"></script>
...
</html>
```

The following example is similar to the previous example except that the script does not specify an output format. Since the first child element of the root node is `<html>`, the output format defaults to HTML.

```
version 1.1;

output-method {
    doctype-public "-//W3C//DTD XHTML 1.0 Transitional//EN";
    doctype-system "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd";
    encoding "utf-8";
    indent "yes";
    omit-xml-declaration "no";
    standalone "no";
    version "1.0";
}

match / {
    <html> {
        <script type="text/javascript" src="/assets/js/api.js">;
     /* ... */
```

```
    }
}
```

The default output format is HTML. The XML declaration is omitted from the output.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html><script type="text/javascript" src="/assets/js/api.js"></script></html>
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

param

IN THIS SECTION

Syntax

```
param $name=value;
```

Description

Declare a parameter for a template or for the style sheet as a whole. Template parameters declared with the `param` statement must be placed inside the template code block. A global parameter, the scope of which is the entire style sheet, must be declared at the top level of the style sheet. You can include an initial value by following the parameter name with an equal sign (=) and a value expression. A parameter whose value is set by Junos OS at script initialization must be defined as a global parameter.

In SLAX, parameter and variable names are declared and accessed using the dollar sign ($). This is unlike the `name` attribute of `<xsl:variable>` and `<xsl:parameter>` elements, which do not include the dollar sign in the declaration.

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

- "Example: Impose a Minimum MTU Setting" on page 732
- "Example: Limit the Number of ATM Virtual Circuits" on page 736
- "Example: Limit the Number of E1 Interfaces" on page 742
- "Example: Prevent Import of the Full Routing Table" on page 781
- "Example: Require and Restrict Configuration Statements" on page 790

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

SLAX Parameters Overview | 104
SLAX Templates Overview | 97
template | 203
var | 211
with | 216

preserve-space

IN THIS SECTION

Syntax

```
preserve-space element-list;
```

Description

Preserve whitespace-only child text nodes for the source tree element nodes listed, but not for the child text nodes of the element node children. To preserve whitespace-only child text nodes of the element node children, specify the child nodes as separate entries in the preserve-space element list. Specifying an asterisk preserves whitespace-only child elements for all elements, which is the default behavior. A text node is considered whitespace-only if it includes only spaces, tabs, newlines, and carriage returns.

The `preserve-space` statement is only needed if the `strip-space` statement has been used with an asterisk, indicating that whitespace-only child text nodes should be removed from all element nodes. In this case, use the `preserve-space` statement to indicate specific element nodes that should not have their whitespace-only child text nodes stripped.

This statement must be defined as a top-level statement in the script.

Attributes

`element-list` Space-separated list of element names for which to preserve whitespace-only child text nodes.

SLAX Example

The following example removes all whitespace-only text nodes from the source tree except for child elements of `<user-context>`:

```
version 1.1;

preserve-space user-context;
strip-space *;

match / {
...
}
```

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

strip-space | 201

priority

IN THIS SECTION

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.

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

apply-templates | 131

call | 138

match | 170

mode | 173

template | 203

with | 216

xsl:template | 70

processing-instruction

IN THIS SECTION

Syntax

```
processing-instruction instruction-name;
```

```
processing-instruction instruction-name {
   instruction-value;
}
```

Description

Add an XML processing instruction to the result tree. A processing instruction is a mechanism to convey application-specific information inside an XML document. The application can detect processing instructions and change their behavior accordingly. The instruction name is mandatory and becomes the target of the processing instruction. It can be a hard-coded string, a variable, or an XPath expression. The optional body generates the processing instruction's content, which consists of one or more name-value pairs. The generated instruction is enclosed within the tags `<?` and `?>`.

Junos OS SLAX scripts generally do not require the `processing-instruction` statement, because the result tree is processed directly by Junos OS. However, you might add a processing instruction to an XML document that is written to disk through the `<xsl:document>` instruction element or one of its related extension elements.

Attributes

instruction-name	Identifier for the processing instruction, which can be a string, a variable, or an XPath expression.
instruction-value	Instruction content, which consists of name-value pairs.

SLAX Example

The following code creates the processing instruction xml-stylesheet. The instruction content contains two name-value pairs: type and href.

```
processing-instruction "xml-stylesheet" {
    expr 'type="text/css" ';
    expr 'href="style.css"';
}
```

The corresponding output in the result tree is:

```
<?xml-stylesheet type="text/css" href="style.css"?>
```

The following example writes an XML document to the file **/var/tmp/output.xml** using the <xsl:document> instruction element. The script adds a processing instruction named instruction to the document.

```
version 1.1;

match / {
    <op-script-results> {
        <xsl:document href="/var/tmp/output.xml" indent="yes" method="xml"> {
            <document-element> {
                <element>;
                processing-instruction "instruction" {
                    expr 'name="testing"';
                }
                <element>;
            }
        }
    }
}
```

The script generates the file **/var/tmp/output.xml**, which contains the processing instruction enclosed within <? and ?> tags.

```
<?xml version="1.0"?>
<document-element>
    <element/>
```

```
    <?instruction name="testing"?>
    <element/>
</document-element>
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

result

IN THIS SECTION

Syntax

```
result expression;
```

```
result {
  /* body */
}
```

Description

Define the return value for a function. The value can be a simple scalar value, an XML element or XPath expression, or a set of instructions that emit the value to be returned.

Attributes

`result` ***`expression`*** XPath expression defining the return value of the function.

SLAX Example

The following example defines three extension functions, `my:size()`, `my:box-parts()`, and `my:ark()`. The `my:ark()` function returns a node set containing an `<ark>` element that encloses the node set returned by the `my:box-parts()` function. The `my:box-parts()` function returns a node set containing a `<box>` element enclosing three `<part>` child elements. The content of each `<part>` element is the value returned by the `my:size()` function. The return value of the `my:size()` function is the product of the three parameters `width`, `height`, and `scale`.

```
version 1.1;
ns my  exclude = "http://www.example.com/myfunctions";

function my:size ($x, $y, $scale = 1) {
    result $x * $y * $scale;
}
function my:box-parts ($width, $height, $depth, $scale = 1) {
    result <box> {
        <part count=2> my:size($width, $depth);
        <part count=2> my:size($width, $height);
        <part count=2> my:size($depth, $height);
    }
}
function my:ark () {
    result {
        <ark> {
            copy-of my:box-parts(2.5, 1.5, 1.5);
        }
    }
}

match / {
    var $res = my:ark();
    copy-of $res;
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

set

IN THIS SECTION

Syntax

```
set $name = value;
```

Description

Assign a value to a mutable variable. The variable must be defined using the `mvar` statement.

Attributes

name Name of the mutable variable.

value Value to assign to the mutable variable.

SLAX Example

The following example creates the mutable variable, `block`. The `set` statement assigns an initial value to the `block` variable. In the `for` loop, the code iterates over each item in the specified list and appends an `<item>` element with two child elements, `<name>` and `<size>`, to the node set stored in the `block` variable.

```
mvar $block;
set $block = <block> "item list";

for $item (list) {
    append $block += <item> {
        <name> $item/name;
        <size> $item/size;
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

SLAX Variables Overview | 110
append | 127
mvar | 175
var | 211

sort

IN THIS SECTION

Syntax

```
sort expression {;
    case-order "upper-first" | "lower-first";
    data-type "text" | "number" | type-name;
    order "ascending" | "descending";
}
```

Description

Control the order in which the `for-each` and `apply-templates` statements iterate through the current node list. By default, the `for-each` and `apply-templates` statements consider nodes in document order, but the `sort` statement defines the order prior to iterating through the node list. Insert the `sort` statement immediately after the `for-each` or `apply-templates` statement. The `sort` statement is only processed when the loop is first initiated.

The `sort` statement has an optional XPath expression and three optional parameters: `case-order`, `data-type`, and `order`. The XPath expression determines each node's comparison string used for sorting. The script evaluates the expression with the node as its context, and then translates the result into the comparison string for that node. If you do not specify an XPath expression, the default value is ".", which causes the string content of each node in the list to be compared. SLAX-specific operators such as == and _ cannot be used within the expression string. If the `sort` statement does not include any optional parameters, the list is sorted based on the string value of each node.

The `sort` statement does not permanently sort the underlying XML data structure, only the order of the current node list being used by the `for-each` or `apply-templates` statement. Multiple `sort` statements can be assigned to a single `for-each` or `apply-templates` statement. They are applied, in order, until a difference is found.

Attributes

`expression` XPath expression that determines each node's comparison string used for sorting. The default value is ".".

`case-order` Specify whether to sort lowercase first or uppercase first. Acceptable values are “lower-first” or “upper-first”. The default is “upper-first”.

`data-type` Specify the element type, which determines whether a numerical, lexical, or other sort is performed. Acceptable values are “number” and “text”. The default is "text".

Setting `data-type` to “text” compares the strings based on their character values (that is ASCII code), so “0” is less than “9”, which is less than “A”, which is less than “Z”, which is less than “a”, which is less than “z”. Setting `data-type` to “number” converts the strings to numbers and compares them numerically. With ascending text sorting, “100” would come before “11” because “0” has a lower ASCII code than “1”, but with ascending number sorting, 11 would come before 100 because 11 is a smaller number than 100.

`order` Specify whether to sort in ascending or descending order. Acceptable values are “descending” or “ascending”. The default is “ascending”.

SLAX Example

The following example SLAX script executes the Junos XML API `get-interface-information` command and parses the resulting output. The `for-each` loop prints the name of each physical interface on the device sorted in ascending order.

```
version 1.1;
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";

match / {
    <op-script-results> {

        var $results = jcs:invoke("get-interface-information");
        for-each ($results/physical-interface/name) {
            sort . {
                data-type "text";
```

```
                order "ascending";
            }
            <interface-name> .;
        }
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

apply-templates | 131
for-each | 158

strip-space

IN THIS SECTION

Syntax

```
strip-space element-list;
```

Description

Remove whitespace-only child text nodes from the source tree element nodes listed, but not from the child text nodes of the element node children. To perform whitespace stripping on the child text nodes of the element node children, specify the child nodes as separate entries in the strip-space element list. Specifying an asterisk removes whitespace-only child elements from all elements. A text node is considered whitespace-only if it includes only spaces, tabs, newlines, and carriage returns.

This statement must be defined as a top-level statement in the script. The default is to preserve all whitespace-only elements.

Attributes

element-list	List of element names separated by spaces.

SLAX Example

The following example removes all whitespace-only text nodes from the source tree except for child elements of `<user-context>`:

```
version 1.1;

preserve-space user-context;
strip-space *;

match / {
...
}
```

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

preserve-space | 189

template

IN THIS SECTION

Syntax

```
template qualified-name (parameter-name = value) {
    /* code */
}
```

Description

Declare a named 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>
```

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

SLAX Parameters Overview | 104
SLAX Templates Overview | 97
apply-templates | 131
call | 138
match | 170

terminate

IN THIS SECTION

Syntax

```
terminate expression;
```

```
terminate {
   /* body */
}
```

Description

Generate an error message that is immediately displayed to the user, and exit the script.

Junos OS op scripts, event scripts, and commit scripts prepend "error:" to the displayed message when generating text output. When generating XML output, the scripts place the output inside a `<message>` element, which is enclosed in an `<xmn:error>` element.

If the `terminate` statement is used in a commit script, the script will generate two errors and terminate the script and the commit process. If the `terminate` statement is used in an event script, the script writes the message to the output file, if one is configured, and terminates the script.

Attributes

expression XPath expression or string emitted as output.

SLAX Example

```
if (not(valid)) {
    terminate "The " _ name() _ " node is not valid. Exiting script.";
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

message | 172

trace

IN THIS SECTION

Syntax

```
trace expression;

trace {
   /* body */
}
```

Description

Write a message to the trace file, if tracing is enabled. If tracing is not enabled, trace output is not generated. The `trace` message can be an XPath expression or string, or it can be generated by the contents of a `trace` statement block.

Enabling of tracing is typically a feature of the environment in which a SLAX script is called. When executing a script using the `slaxproc` command, include the `--trace` or `-t` option to enable tracing. For more information about slaxproc, see "Understanding the SLAX Processor (slaxproc)" on page 247.

Attributes

`trace` ***`expression`***	XPath expression or string written to the trace file.

SLAX Example

The following examples demonstrate the `trace` statement syntax. The first example writes a concatenated string to the trace file. The second example uses a code block to output a `<max>` element and a `<min>` element and the values of the `max` and `min` variables. The third example uses a conditional statement to specify when to output trace data. If the expression evaluates to `true`, the code block writes the string and the `<options>` element hierarchy to the trace file.

```
trace "max " _ $max _ "; min " _ $min;

trace {
    <max> $max;
    <min> $min;
}

trace {
    if ($my-trace-flag) {
```

```
        expr "max " _ $max _ "; min " _ $min;
        copy-of options;
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

message | 172

terminate | 205

Understanding the SLAX Processor (slaxproc) | 247

uexpr

IN THIS SECTION

Syntax

```
uexpr expression;
```

Description

Generate the string value of an XPath expression and add it to the result tree, but do not escape special characters. The `uexpr` statement behaves identically to the `expr` statement, except that the contents are not escaped. By default, characters such as "<", ">", and "&" are escaped into proper XML as "<", ">", and "&", respectively, but `uexpr` does not execute this escaping mechanism.

Attributes

expression	XPath expression to add to the result tree.

SLAX Example

The following statement outputs the string to the result tree exactly as it appears in the statement. If `expr` is used in place of `uexpr`, the script would output the string "<:&>".

```
uexpr "<:-&>";
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

expr | 152
message | 172
terminate | 205
trace | 206

use-attribute-sets

IN THIS SECTION

Syntax

```
use-attribute-sets attribute-set-name;
```

Description

Add the attributes in the attribute set to the current element. The `use-attribute-sets` statement can be used under the `attribute-set`, `copy-node`, and `element` statements, as well as under a normal element.

Attributes

attribute-set-name	Name of the attribute set, which is defined using an `attribute-set` statement.

SLAX Example

The following example creates two attribute sets: `table-attributes` and `table-attributes-ext`. The `table-attributes-ext` set includes all of the attributes that are already defined in the `table-attributes` set through use of the `use-attributes-sets` statement. In the main script body, the `table-attributes-ext` attribute set is applied to the `<table>` element. The `<table>` element includes the four attributes: `order`, `cellpadding`, `cellspacing`, and `border`.

```
version 1.1;

var $cellpadding = "0";
```

```
var $cellspacing = "10";

attribute-set table-attributes {
    attribute "order" { expr "0"; }
    attribute "cellpadding" { expr $cellpadding; }
    attribute "cellspacing" { expr $cellspacing; }
}
attribute-set table-attributes-ext {
    use-attribute-sets table-attributes;
    attribute "border" { expr "0"; }
}

match / {
    ...
    <table> {
        use-attribute-sets table-attributes-ext;
    }
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

SLAX Elements and Element Attributes Overview | 89
attribute | 133
attribute-set | 135
element | 146

var

IN THIS SECTION

Syntax

```
var $name=value;
```

Description

Declare a local or global variable. A variable is global if it is defined outside of any template. Otherwise, it is local. The value of a global variable is accessible anywhere in the style sheet. The scope of a local variable is limited to the template or code block in which it is defined. Variables declared in this manner are immutable. You initialize a variable by following the variable name with an equal sign (=) and an expression.

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: Limit the Number of E1 Interfaces" on page 742, "Example: Limit the Number of ATM Virtual Circuits" on page 736, "Example: Configure Administrative Groups for LSPs" on page 684, and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

SLAX Variables Overview | 110
mvar | 175
param | 187

version

IN THIS SECTION

Syntax

```
version version-number;
```

Description

Specify the version of SLAX that is being used. All SLAX style sheets must begin with a `version` statement.

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. Acceptable values are 1.0, 1.1, and 1.2.

SLAX Example

```
version 1.0;
```

XSLT Equivalent

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

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

Required Boilerplate for Commit Scripts | 542

Required Boilerplate for Event Scripts | 1084

Required Boilerplate for Op Scripts | 822

SLAX Syntax Rules Overview | 86

while

IN THIS SECTION

Syntax

```
while (expression) {
    /* body */
}
```

Description

Repeatedly execute a block of statements until the specified condition evaluates to `false`. The condition is an XPath expression that is converted to a boolean type. If the expression evaluates to `true`, the contents of the while loop are executed. The loop continues to execute until the expression evaluates to `false`. During execution, the context is not changed. In the expression, you should use a mutable variable, which is declared using the `mvar` statement, to avoid creating an infinite loop.

Attributes

`expression` XPath expression, which is cast to boolean type and used as the condition for the while loop. The code block contents are executed as long as the condition evaluates to `true`.

SLAX Example

In the example, the while loop parses through the item list until the desired value is found. When that value is detected, `$seen` is set to true, and the while loop exits.

```
mvar $seen = false();
mvar $count = 1;

while (not($seen)) {
    if (item[$count]/value) {
        set $seen = true();
    }
    set $count = $count + 1;
}
```

Release Information

Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

SLAX Variables Overview | 110
XPath Overview | 22
mvar | 175

with

IN THIS SECTION

Syntax

```
with $name = value;
```

Description

Specify a parameter to pass into a template. You can use this statement when you apply templates with the `apply-templates` statement or invoke templates with the `call` statement.

Optionally, you can specify a value for the parameter by including an equal sign (=) and a value expression. If no value is specified, the current value of the parameter is passed to the template.

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: Configure Dual Routing Engines" on page 696, "Example: Prevent Import of the Full Routing Table" on page 781, and "Example: Automatically Configure Logical Interfaces and IP Addresses" on page 674.

Release Information

Statement introduced in version 1.0 of the SLAX language.

RELATED DOCUMENTATION

SLAX Parameters Overview | 104

SLAX Templates Overview | 97

apply-templates | 131

call | 138

match | 170

CHAPTER 8

The libslax Distribution for Automation Scripting

IN THIS CHAPTER

libslax Distribution Overview

Stylesheet Language Alternative syntaX (SLAX) is an alternative syntax for XSLT and is tailored for readability and familiarity, following the style of C and Perl. In the SLAX language, programming constructs and XPath expressions are moved from the XML elements and attributes used in XSLT to first class language constructs. SLAX was originally developed as part of Junos OS for the purpose of on-box scripting to enable users to customize and enhance the command-line interface (CLI).

libslax is an open-source implementation of the SLAX language using the "New BSD License." libslax is written in C and is built on top of the libxml2, libxslt, and libexslt libraries. The libslax distribution contains the libslax library, which incorporates a SLAX writer and SLAX parser, a debugger, a profiler, and the SLAX processor (slaxproc). The SLAX processor is a command-line tool that can validate SLAX script syntax, convert between SLAX and XSLT formats, and format, debug, or run SLAX scripts.

You can download and install the libslax distribution on a computer with a UNIX-like operating system to develop SLAX scripts outside of Junos OS. The libslax tools are also included as part of the standard Junos OS. Table 8 on page 221 outlines the libslax version included with each Junos OS release.

Table 8: libslax Version on Devices Running Junos OS

Junos OS Release	libslax Version	SLAX Version
12.2 through 13.3	0.11.22	1.1
14.1	0.14.1	1.1
14.2 through 17.3	0.17.1	1.2
17.4 and later	0.22.0	1.2

For more information about libslax, including source code, release notes, additional documentation, and support materials, see the following sites:

- https://github.com/Juniper/libslax
- https://github.com/Juniper/libslax/wiki
- http://juniper.github.io/libslax/slax-manual.html

RELATED DOCUMENTATION

Download and Install the libslax Distribution | 223

SLAX Overview | 83

Understanding the SLAX Processor (slaxproc) | 247

libslax Library and Extension Libraries Overview | 221

How to Use the SLAX Processor (slaxproc) | 254

libslax Library and Extension Libraries Overview

IN THIS SECTION

libslax Library

libslax is an open-source implementation of the SLAX language using the "New BSD License." libslax is written in C and is built on top of the libxml2, libxslt, and libexslt libraries.

The core of the libslax distribution is the libslax library, which incorporates a SLAX parser to read SLAX files, a SLAX writer to write SLAX files, a debugger, a profiler, and the SLAX processor (slaxproc) command-line tool . The parser turns a SLAX source file into an XSLT tree (xmlDocPtr) using the `xsltSetLoaderFunc()` hook. The writer turns an XSLT tree (xmlDocPtr) into a file containing SLAX statements.

libslax Extension Libraries

libslax provides supports for dynamically loading extension libraries. The libslax distribution includes the `bit`, `curl`, `db`, `os`, and `xutil` extension libraries. The source files for the default extension libraries are stored in the **libslax-*release*/extensions** directory of the distribution. You can supply additional extension libraries beyond the default extension libraries supported by the libslax distribution. Extension library locations can be specified statically at build time or dynamically at runtime.

By default, libslax installs architecture-independent files, including extension library files, in the **/usr/local** directories. Specifically, libslax installs the extension libraries in the **/usr/local/lib/slax/extensions** directory. If you do not specify a different installation directory for the extension libraries at build time, the SLAX processor checks this directory for extension libraries when executing a script.

There are several ways to specify extension library locations at build time. During installation, to specify a directory prefix other than **/usr/local** for all installation files, including the libraries, execute the **./configure** command and include the `--prefix=`*`prefix`* option specifying the location to install the files. The default extension libraries are installed in the ***prefix*/lib/slax/extensions** directory, and the SLAX processor checks this directory for extension libraries when executing a script. To install just the extension library files in a different, user-defined location, execute the **./configure** command and include the `--with-extensions-dir=`*`dir`* option specifying the location where the extension libraries live. The SLAX processor will then automatically check the specified directory for extension libraries when executing a script. For more information about installing libslax, see "Download and Install the libslax Distribution" on page 223.

There are several ways to specify extension library locations dynamically after installation is complete. You can define or update the SLAXEXTPATH environment variable to include the directory locations of additional extension libraries. The variable value is a colon-separated list of directories. The SLAX processor automatically checks these directories for extension libraries when executing a script. Alternatively, you can specify the extension library location when you execute a script by using the `slaxproc` command with the `--lib` or `-L` option.

To summarize, extension library locations are supplied to the SLAX processor in one of the following ways:

- By default, in the **/usr/local/lib/slax/extensions** directory.
- In **lib/slax/extensions/** under the directory specified by the `./configure --prefix` option given at build time.
- In the user-defined directory specified by the `./configure --with-extension-dir` option given at build time.
- In a directory included in the colon-separated list of the SLAXEXTPATH environment variable.
- In a directory provided using the `--lib` or `-L` argument to the `slaxproc` command.

RELATED DOCUMENTATION

libslax Default Extension Libraries: bit, curl, db, os, and xutil | 224
Download and Install the libslax Distribution | 223
libslax Distribution Overview | 220

Download and Install the libslax Distribution

The libslax distribution contains the libslax library, which incorporates a SLAX writer and SLAX parser, a debugger, a profiler, and the SLAX processor (slaxproc). The SLAX processor is a command-line tool that can validate SLAX script syntax, convert between SLAX and XSLT formats, and format, debug, or run SLAX scripts.

The libslax tools are included as part of the standard Junos OS. However, you can download and install the libslax distribution on a computer with a UNIX-like operating system to develop SLAX scripts outside of Junos OS.

To download and install the libslax distribution, follow the instructions on the libslax wiki.

https://github.com/Juniper/libslax/wiki/Building

Review the **INSTALL** file that comes with the distribution for additional information.

RELATED DOCUMENTATION

libslax Distribution Overview | 220
libslax Library and Extension Libraries Overview | 221
Understanding the SLAX Processor (slaxproc) | 247

libslax Default Extension Libraries: bit, curl, db, os, and xutil

IN THIS SECTION

libslax bit Extension Library

The libslax **bit** extension library contains functions that create and manipulate bit strings. The functions support 64-bit integer arguments. To incorporate functions from the **bit** extension library into SLAX scripts, include the namespace statement for that library in the script.

```
ns bit extension = "http://xml.libslax.org/bit";
```

Call the bit extension functions using the `bit` prefix and the function name. For example:

```
version 1.1;
ns bit extension = "http://xml.libslax.org/bit";

var $a = 63;
var $b = { expr "10111"; }

match / {
    <out> {
        <bit-and> {
            <a1> bit:and("101100", "100101");
            <a2> bit:and($a, $b);
            <a3> bit:and($a, number($b));
        }
        <bit-or> {
            <a1> bit:or("101100", "100101");
            <a2> bit:or($a, $b);
```

```
            <a3> bit:or($a, number($b));
        }
        <bit-mask> {
            <a1> bit:mask(0);
            <a2> bit:mask(8, 32);
        }
        <ops> {
            <a1> bit:to-int("10101");
        }
    }
}
```

Table 9 on page 225 lists the functions available in the **bit** extension library, which is supported in SLAX 1.1 scripts in Junos OS Release 12.2 and later.

Table 9: Functions in the libslax bit Extension Library

Function and Arguments	Description	Example
`bit:and(b1, b2)`	Return the logical AND of two bit strings.	`bit:and("101100", "100101")` return value: "100100"

`bit:clear(b1,bitnum)`	Set the specified bit in the bit string to zero and return the new bit string. Bits are numbered starting from zero. If the integer argument is greater than the bit string length, the bit string is extended.	`bit:clear("11111", 0)`
		return value: "11110"
		
		`bit:clear("11111", 6)`
		return value: "0011111"

`bit:compare(value1, value2)`	Compare two values and return an integer less than, equal to, or greater than zero, if the first argument is found to be less than, equal to, or greater than the second argument, respectively.	`bit:compare("10000", 16)`
		return value: 0
		
		`bit:compare("11111", "10000")`
		return value: 1

`bit:from-hex(string, len?)`	Return the value of the hexademical argument as a bit string. The optional second argument pads the bit string with leading zeros (0s) until it is the specified length.	`bit:from-hex("0x45", 8)` return value: "01000101"

Table 9: Functions in the libslax bit Extension Library *(Continued)*

Function and Arguments	Description	Example
`bit:from-int(integer, len?)`	Return the value of the integer argument as a bit string. The optional second argument pads the bit string with leading zeros until it is the specified length.	`bit:from-int(65,8)` return value: "01000001"
`bit:mask(count, len?)`	Return a bit string with the specified number of low-order bits set to one. The optional second argument pads the bit string with leading zeros until it is the specified length.	`bit:mask(4, 8)` return value: "00001111"
`bit:nand(b1, b2)`	Return the logical NAND of two bit strings.	`bit:nand("101100", "100101")` return value: "010010"
`bit:nor(b1, b2)`	Return the logical NOR of two bit strings.	`bit:nor("101100", "100101")` return value: "011011"
`bit:not(b1)`	Return the inversion (NOT) of a bit string.	`bit:not("101100")` return value: "010011"
`bit:or(b1, b2)`	Return the logical OR of two bit strings.	`bit:or("101100", "100101")` return value: "101101"

`bit:set(b1, bitnum)`	Set the specified bit in the bit string and return the new bit string. Bits are numbered starting from zero. If the integer argument is greater than the bit string length, the bit string is extended.	`bit:set("1001", 2)`
		return value: "1101"
		
		`bit:set("1001", 6)`
		return value: "1001001"

`bit:to-int(b1)`	Return the value of the bit string argument as an integer.	`bit:to-int("101100")` return value: 44
`bit:to-hex(b1)`	Return the value of the bit string argument as a string representation of the hexadecimal value.	`bit:to-hex("101100")` return value: "0x2c"

Table 9: Functions in the libslax bit Extension Library *(Continued)*

Function and Arguments	Description	Example
`bit:xor(b1, b2)`	Return the logical XOR of two bit strings.	`bit:xor("101100", "100101")` return value: "001001"
`bit:xnor(b1, b2)`	Return the logical XNOR of two bit strings.	`bit:xnor("101100",  "100101")` return value: "110110"

libslax curl Extension Library

IN THIS SECTION

Understanding the libslax curl Extension Library

cURL is a command-line tool that uses the libcurl library and supports data transfers using a number of protocols, including FTP, FTPS, HTTP, HTTPS, SCP, and SMTP. For more information about cURL, see the cURL website at http://curl.haxx.se/.

The libslax distribution includes the **curl** extension library, which contains functions that perform cURL operations. To incorporate functions from the libslax **curl** extension library in SLAX scripts, include the namespace statement for that library in the script.

```
ns curl extension = "http://xml.libslax.org/curl";
```

You can then call the **curl** extension functions using the `curl` prefix and the function name. cURL operations are directed using a set of elements passed to the extension functions.

Table 10 on page 228 lists the supported operations in the **curl** extension library and includes a description of each function. Table 11 on page 228 and Table 12 on page 230 list the supported elements and include the syntax and a description of each element. More specifically, Table 11 on page 228 lists elements used for web services operations, and Table 12 on page 230 lists the elements used for e-mail operations. For examples and additional information about using the **curl** extension library, see The curl Extension Library.

Table 10: Functions in the libslax curl Extension Library

Function	Description
`curl:close`	Close an open connection. Further operations cannot be performed over the connection. See "curl:close" on page 231.
`curl:open`	Open a connection to a remote server, allowing multiple operations over a single connection. See "curl:open" on page 231.
`curl:perform`	Perform simple transfers using a persistent connection handle provided by `curl:open`. See "curl:perform" on page 232.
`curl:set`	Record a set of parameters that persists for the lifespan of a connection. See "curl:set" on page 233.
`curl:single`	Perform transfer operations without using a persistent connection. See "curl:single" on page 234.

Table 11: Web Services Elements in the libslax curl Extension Library

Element	Description	Syntax
`<connect-timeout>`	Number of seconds before a connection attempt is considered to have failed.	`<connect-timeout> 10;`
`<content-type>`	MIME type for the transfer payload.	`<content-type> "mime/type";`

Table 11: Web Services Elements in the libslax curl Extension Library *(Continued)*

Element	Description	Syntax
`<errors>`	Control how HTML and XML parsing errors are handled.	See "Handling Errors" on page 239
`<fail-on-error>`	Indicate that the transfer should fail if any errors, including insignificant ones, are detected.	`<fail-on-error>;`
`<format>`	Specify the expected format of returned results, allowing the **curl** extension to automatically make the content available in the native format. Formats include "html", "name", "text", "url-encoded", and "xml".	`<format> "xml";`
`<header>`	Provide additional header fields for the request.	`<header name="name"> "value";`
`<insecure>`	Indicate a willingness to tolerate insecure communications operations. Specifically, allow SSL Certs without checking the common name.	`<insecure>;`
`<method>`	Method used to transfer data. This controls the HTTP request type, as well as triggering other transfer mechanisms. Acceptable method names include "get", "post", "delete", "head", "email", "put", and "upload". "get" is the default.	`<method> "get";`
`<param>`	Provide additional parameter values for the request. These parameters are typically encoded into the URL.	`<param name="x"> "y";`
`<password>`	User's password to use for the transfer.	`<password> "password";`
`<secure>`	Request the use of the secure version of a protocol, including HTTPS and FTPS.	`<secure>;`
`<timeout>`	Number of seconds before an open connection is considered to have failed.	`<timeout> 10;`

Table 11: Web Services Elements in the libslax curl Extension Library *(Continued)*

Element	Description	Syntax
`<upload>`	Indicate this is a file upload request.	`<upload>;`
`<url>`	Base URL for the request.	`<url> "target-url";`
`<username>`	Username to use for the transfer.	`<username> "username";`
`<verbose>`	Request detailed debug information about the operations and communication of the transfer.	`<verbose>;`

Table 12: E-Mail Elements in the libslax curl Extension Library

Element	Description	Syntax
`<cc>`	"Cc" address for e-mail (SMTP) requests. For multiple addresses, use multiple <cc> elements.	`<cc> "cc-user@email.example.com";`
`<contents>`	Contents to be transferred.	`<contents> "multi-\nline\ncontents\n";`
`<from>`	"From" address for e-mail (SMTP) requests.	`<from> "source-user@email.example.com";`
`<local>`	Local hostname used for e-mail (SMTP) requests.	`<local> "local host name";`
`<server>`	Outgoing SMTP server name. Currently, MX records are not processed.	`<server> "email-server.example.com";`
`<subject>`	"Subject" field for e-mail (SMTP) requests.	`<subject> "email subject string";`
`<to>`	"To" address for e-mail (SMTP) requests. For multiple addresses, use multiple <to> elements.	`<to> "to-user@email.example.com";`

The libcurl elements closely mimic the options used by the native C libcurl API in libcurl's `curl_easy_setopt()` function. Once the options are set, a call to `curl_easy_perform()` performs the requested transfer. For more information about the `curl_easy_setopt()` function, see http://curl.haxx.se/libcurl/c/curl_easy_setopt.html .

In the libslax **curl** extension library, the libcurl API options are represented as individual elements. For example, the `<url>` element is mapped to the CURLOPT_URL option , the `<method>` element is mapped to the CURLOPT_CUSTOMREQUEST option, and so forth.

These elements can be used in three ways:

- The `curl:single()` extension function supports using a set of options in a single transfer operation with no persistent connection handle.
- The `curl:perform()` extension function supports using a set of options with a persistent connection handle. The handle is returned from the `curl:open()` extension function and can be closed with the `curl:close()` extension function.
- The `curl:set()` extension function records a set of options for a connection handle and keeps those options active for the lifetime of the connection. For example, if the script needs to transfer a number of files, it can record the `<username>` and `<password>` options and avoid repeating them in every `curl:perform()` call.

curl:close

The `curl:close()` extension function closes an open connection. Further operations cannot be performed over the connection once it is closed.

The syntax is:

```
node-set[empty] curl:close(node-set[connection]);
```

The argument is the connection handle to close.

curl:open

The `curl:open()` extension function opens a connection to a remote server, allowing multiple operations over a single connection.

The syntax is:

```
node-set[connection] curl:open();
```

The returned object is a connection handle that can be passed to `curl:perform()` or `curl:close()`.

curl:perform

The `curl:perform()` extension function performs simple transfers using a persistent connection handle provided by `curl:open()`.

The syntax is:

```
node-set[object] curl:perform(node-set[connection], node-set[options])
```

The arguments are the connection handle and a set of option elements. Supported **curl** extension library elements are defined in Table 11 on page 228 and Table 12 on page 230.

The returned object is an XML hierarchy containing the results of the transfer. Table 13 on page 232 lists the possible elements in the reply, and Table 14 on page 233 lists the possible elements contained within the `<header>` element.

Table 13: curl:perform Reply Elements

Element	Contents
`<curl-success>`	Empty element which Indicates success
`<data>`	Parsed data
`<error>`	Error message text, if any
`<header>`	Parsed header fields
`<raw-data>`	Raw data from the reply
`<raw-headers>`	Raw header fields from the reply
`<url>`	Requested URL

Table 14: curl:perform <header> Elements

Element	Contents
`<code>`	HTTP reply code
`<field>`	HTTP reply field (with @name and value)
`<message>`	HTTP reply message
`<version>`	HTTP reply version string

The following example shows the `<header>` element with header fields parsed into `<field>` elements:

```
<header>
    <version>HTTP/1.1</version>
    <code>404</code>
    <message>Not Found</message>
    <field name="Content-Type">text/html</field>
    <field name="Content-Length">345</field>
    <field name="Date">Mon, 08 Aug 2011 03:40:21 GMT</field>
    <field name="Server">lighttpd/1.4.28 juisebox</field>
</header>
```

curl:set

The `curl:set()` extension function records a set of parameters that persist for the lifespan of a connection.

The syntax is:

```
node-set[empty] curl:set(node-set[handle], node-set[options]);
```

The arguments are the connection handle and a set of option elements. Supported **curl** extension library elements are defined in Table 11 on page 228 and Table 12 on page 230.

curl:single

The `curl:single()` extension function performs transfer operations without using a persistent connection.

The syntax is:

```
node-set[result] curl:single(node-set[options]);
```

The returned object is identical in structure to the one returned by `curl:perform()`. Refer to "curl:perform" on page 232 for additional information.

Supported Format Elements

Starting with version 1.2 of the SLAX language, which is supported in Junos OS Release 14.2 and later releases, the `format` element supports two new formats in addition to "html", "text", and "xml": "name" and "url-encoded". The "name" encoding is used for name=value pairs that are separated by newlines, whereas the "url-encoded" encoding is used when the name=value pairs are separated by an ampersand (&).

The parsed data is returned in the `<data>` element, using `<name>` elements. In the following example, `format` is set to "url-encoded":

```
<results>
   <url>http://api.example.com/request_token</url>
   <curl-success/>
   <raw-headers>HTTP/1.1 200 OK&#xD;
 Server: XXXX&#xD;
 Date: Tue, 18 Jun 2013 18:56:31 GMT&#xD;
 Content-Type: application/x-www-form-urlencoded&#xD;
 Transfer-Encoding: chunked&#xD;
 Connection: keep-alive&#xD;
 x-server-response-time: 69&#xD;
 x-example-request-id: 123456&#xD;
 pragma: no-cache&#xD;
 cache-control: no-cache&#xD;
 x-http-protocol: None&#xD;
 x-frame-options: SAMEORIGIN&#xD;
 X-RequestId: 12345&#xD;
 &#xD;
 </raw-headers>
   <headers>
     <version>HTTP/1.1</version>
```

```
        <code>200</code>
        <message>OK</message>
        <header name="Server">XXXXX</header>
        <header name="Date">Tue, 18 Jun 2013 18:56:31 GMT</header>
        <header name="Content-Type"
               >application/x-www-form-urlencoded</header>
        <header name="Transfer-Encoding">chunked</header>
        <header name="Connection">keep-alive</header>
        <header name="x-server-response-time">69</header>
        <header name="x-example-request-id">123456</header>
        <header name="pragma">no-cache</header>
        <header name="cache-control">no-cache</header>
        <header name="x-http-protocol">None</header>
        <header name="x-frame-options">SAMEORIGIN</header>
        <header name="X-RequestId">12345</header>
      </headers>
      <raw-data>oauth_token_secret=s&amp;oauth_token=t</raw-data>
      <data format="url-encoded">
        <name name="oauth_token_secret">s</name>
        <name name="oauth_token">t</name>
      </data>
    </results>
```

curl Examples

The following examples show SLAX scripts that use the libslax `curl` extension library functions to perform operations.

The following SLAX script performs a simple GET operation to retrieve a web page. The script specifies the header field for the HTTP header and a parameter that is incorporated into the requested URL.

```
version 1.1;

ns curl extension = "http://xml.libslax.org/curl";

param $url = "http://www.juniper.net";

match / {
    <op-script-results> {
        var $options = {
            <header name="client"> "slaxproc";
            <param name="smokey"> "bandit";
```

```
        }
        var $results = curl:single($url, $options);
        message "completed: " _ $results/headers/message;
        <curl> {
            copy-of $results;
        }
    }
}
```

The following SLAX script sends an e-mail by way of a server, which is provided as a parameter:

```
version 1.1;

ns curl extension = "http://xml.libslax.org/curl";

param $server;

match / {
    <out> {
        var $info = {
            <method> "email";
            <server> $server;
            <from> "muffin@example.com";
            <to> "phil@example.net";
            <subject> "Testing...";
            <contents> "Hello,
This is an email.
Thanks,
 Phil
";
        }
        var $res = curl:single($info);
        <res> {
            copy-of $res;
        }
    }
}
```

The following SLAX op script uses the provided credentials to connect and log in to a web application:

```
version 1.2;

ns curl extension = "http://xml.libslax.org/curl";

param $url = "http://198.51.100.10:4000/login";
param $username = "admin";
param $password = "password";

var $params := {
     <url> $url;
     <method> "post";
     <insecure>;
     <param name="username"> $username;
     <param name="password"> $password;
}

main <top> {
     var $curl = curl:open();

     var $data = curl:perform($curl, $params);

     copy-of $data;
     expr curl:close($curl);
}
```

You can use Flask to create a simple web application to test the script. Flask, which is available on PyPI, is a lightweight Web Server Gateway Interface (WSGI) web application framework. The following Python script from https://pythonspot.com/login-authentication-with-flask creates a simple web application with a login page.

```
from flask import Flask
from flask import Flask, flash, redirect, render_template, request, session, abort
import os

app = Flask(__name__)

@app.route('/')
def home():
    if not session.get('logged_in'):
```

```
        return render_template('login.html')
    else:
        return "Hello Boss!"

@app.route('/login', methods=['POST'])
def do_admin_login():
    if request.form['password'] == 'password' and request.form['username'] == 'admin':
        session['logged_in'] = True
    else:
        flash('wrong password!')
    return home()

if __name__ == "__main__":
    app.secret_key = os.urandom(12)
    app.run(debug=True,host='0,0.0.0', port=4000)
```

To test the op script, first start the web application script on the remote device.

```
user@nms:~$ python3 app.py

 * Serving Flask app "app" (lazy loading)
...
```

Then enable and run the op script on the device running Junos OS. The results are logged to the **op-script.log** file.

```
user@router> op curl-test
user@router> show log op-script.log | last 100
...
<?xml version="1.0"?>
<top>
<results>
  <url>http://198.51.100.10:4000/login</url>
  <curl-success/>
  <raw-headers>HTTP/1.0 200 OK&#xD;
Content-Type: text/html; charset=utf-8&#xD;
Content-Length: 11&#xD;
Vary: Cookie&#xD;
Set-Cookie: session=eyJsb2dnZWRfaW4iOnRydWV9.XzxdUg.wm2A58ct6twASerlHjuQ64nwaiI; HttpOnly; Path=/
&#xD;
Server: Werkzeug/1.0.1 Python/3.6.9&#xD;
```

```
Date: Tue, 18 Aug 2020 22:59:30 GMT&#xD;
&#xD;
</raw-headers>
  <headers>
    <version>HTTP/1.0</version>
    <code>200</code>
    <message>OK</message>
    <header name="Content-Type">text/html; charset=utf-8</header>
    <header name="Content-Length">11</header>
    <header name="Vary">Cookie</header>
    <header name="Set-
Cookie">session=eyJsb2dnZWRfaW4iOnRydWV9.XzxdUg.wm2A58ct6twASerlHjuQ64nwaiI; HttpOnly; Path=/</
header>
    <header name="Server">Werkzeug/1.0.1 Python/3.6.9</header>
    <header name="Date">Tue, 18 Aug 2020 22:59:30 GMT</header>
  </headers>
  <raw-data>Hello Boss!</raw-data>
</results>Aug 18 15:59:31 end dump
</top>
...
```

Handling Errors

Starting with version 1.2 of the SLAX language, which is supported in Junos OS Release 14.2 and later releases, you can use the `<errors>` element to control the handling of HTML and XML parsing errors. The default behavior is to display errors in the standard error output stream (stderr). By supplying different values for the `<errors>` element, you can also choose to ignore, log, or record the error. Table 15 on page 239 lists the values that you can use for the `errors` element.

Table 15: Error Elements

Value	Special Behavior
default	Errors are displayed on stderr
ignore	Errors are discarded
log	Errors are logged (via slaxLog())

Table 15: Error Elements *(Continued)*

Value	Special Behavior
record	Errors are recorded

When the value of `<errors>` is set to "record", all errors appear in a string under the `<errors>` element in the XML node (as returned, for example by `curl:perform`). If no errors are generated, the `<errors>` element is not present, which can be used as a test for errors.

```
var $opt = {
    <url> $url;
    <format> "html";
    <errors> "record";
}
var $res = curl:single($opts);
if ($res/errors) {
    terminate "failure: " _ $res/errors;
}
```

libslax db Extension Library

The libslax **db** extension library, which is supported in Junos OS Release 17.4R1 and later, provides a way for SLAX scripts to store, manipulate, retrieve and delete data stored in a database. To use functions from the **db** extension library in SLAX scripts, include the namespace statement for that library in the script.

```
ns db extension = "http://xml.libslax.org/db";
```

Table 16 on page 240 summarizes the functions that are available in the **db** extension library. For examples and additional information about the **db** extension functions, see db Extension Functions.

Table 16: Functions in the libslax db Extension Library

Function	Description
`db:close()`	Close the database connection and free all the structures associated with previous operations performed on this handle.

Table 16: Functions in the libslax db Extension Library *(Continued)*

Function	Description
`db:create()`	Create a collection in the database associated with the open database handle using the given field information.
`db:delete()`	Delete instances in a collection that match the given conditions in the database associated with the open database handle.
`db:fetch()`	Fetch a result instance using the cursor returned from a `db:find()` or `db:query` call.
`db:find()`	Return a cursor to the result set for instances matching the given conditions. `db:fetch()` must be used to retrieve each of the result instances.
`db:find-and-fetch()`	Find and read all the instances matching the given conditions.
`db:insert()`	Insert data into a collection in the database associated with the open database handle.
`db:open()`	Open a database connection using the provided options, which include the back-end database engine and the database name.
`db:query()`	Run a custom query.
`db:update()`	Update a set of instances matching the given conditions with the provided instance.

Table 17 on page 241 summarizes the elements that can be used in the options and data of **db** extensions functions. For examples and additional information about **db** elements, see db Elements.

Table 17: Elements in the db Extension Library

Element	Description	Example
`<collection>`	Specify the data collection on which to perform operations. This corresponds to a SQL database table.	`<collection> "employee";`

Table 17: Elements in the db Extension Library *(Continued)*

Element	Description	Example

<condition>	Specify a condition that must be satisfied when operating with data instances from the datastore. This forms the condition used with the WHERE clause when operating with a SQL datastore. Mandatory elements include:
	• <selector>—Name of the field to which this condition applies
	• <operator>–Comparison or logical operator (<, >, <=, >=, =, LIKE, IN, NOT)
	• <value>—Value used with the operator on this field
		<condition> {
		<selector> "id;
		<operator> "in";
		<value> "(1, 10)";
		}

<conditions>	Specify multiple conditions with <and> or <or> as parent nodes.	<conditions> {
		<and> {
		<condition> {
		... /* c1 */
		}
		<condition> {
		... /* c2 */
		}
		<or> {
		<condition> {
		... /* c3 */
		}
		<condition> {
		... /* c4 */
		}
		}
		}
		}

<database>	Specify the name of the database on which to operate.	<database> "test.db";
<engine>	Specify the back-end database engine that is used to store and access data.	<engine> "sqlite";

Table 17: Elements in the db Extension Library *(Continued)*

<table>
<tr><th>Element</th><th>Description</th><th>Example</th></tr>
<tr><td><field></td><td>Define the metadata for each field in the collection.</td><td><pre><field> {
    <name> "name";
    <type> "integer";
    <primary>;
}</pre></td></tr>
<tr><td><fields></td><td>Specify metadata about fields in a collection using <field> child elements.</td><td><pre><fields> {
    <field> {
        ...
    }
}</pre></td></tr>
<tr><td><instance></td><td>Represent a single instance in a collection when inserting or manipulating data in the datastore. The element contains fields and their corresponding values in that record.</td><td><pre><instance> {
    <id> 5;
    <name> "John";
}</pre></td></tr>
<tr><td><instances></td><td>Define multiple instances in a collection when inserting or manipulating data in the datastore.</td><td><pre><instances> {
    <instance> {
        ...
        ...
    }
    ...
}</pre></td></tr>
<tr><td><limit></td><td>Limit the number of instances that a result can contain.</td><td><pre><limit> 10;</pre></td></tr>
</table>

Table 17: Elements in the db Extension Library *(Continued)*

Element	Description	Example
`<result>`	Output node returned from most of the **db** extension functions. This node contains a `<status>` element and can contain one or more `<instance>` elements.	`<result> { <status> { <ok> } }`
`<retrieve>`	Specify only the fields that should appear as part of the result set when querying the datastore.	`<retrieve> { <id>; <name>; }`
`<skip>`	Skip over a specified number of instances in the result set before returning to the user.	`<skip> 5;`
`<sort>`	Specify the fields and the order by which the result set must be sorted. Specify the fields by including the `<by>` element, and specify the sort order by setting the value of `<order>` to "asc" (ascending) or "desc" (descending).	`<sort> { <by> "id"; <by> "age"; <order> "desc"; }`

libslax os Extension Library

The libslax `os` extension library provides a set of functions to invoke operating system-related operations on the local host. To use functions from the `os` extension library in SLAX scripts, include the namespace statement for that library in the script.

```
ns os extension = "http://xml.libslax.org/os";
```

Table 18 on page 245 summarizes the functions that are available in the os extension library. The return value of many `os` extension functions consists of a set of zero or more error nodes. Each node can contain an `<error>` element with additional child elements. For examples and additional information about `os` extension functions, see os Extension Functions.

Table 18: Functions in the libslax os Extension Library

Function	Description	First Supported Junos OS Release
`os:chmod`	Change the permissions of one or more files.	17.4
`os:chown`	Change the file owner and group for one or more files.	17.4
`os:exit-code`	Set the exit code for the process running the script.	14.1
`os:mkdir`	Create a new directory with the specified name, path, and permissions. The `<create>` option, which returns an error if the last element of the path exists, is supported starting in Junos OS Release 17.4R1.	14.1
`os:remove`	Delete a file or empty directory.	17.4
`os:stat`	Return information about files and directories in a node-set of `<entry>` elements, which contain details about each file.	14.1
`os:user-info`	Return a `<user-info>` element with information about the user running the script.	17.4

libslax xutil Extension Library

The libslax **xutil** extension library contains functions that convert between strings and XML node sets. To incorporate functions from the **xutil** extension library into SLAX scripts, include the namespace statement for that library in the script.

```
ns xutil extension = "http://xml.libslax.org/xutil";
```

Call the **xutil** extension functions using the **xutil** prefix and the function name. Table 19 on page 246 lists the functions available in the **xutil** extension library, which are supported in SLAX 1.1 and later scripts. For examples and additional information about **xutil** extension functions, see xutil Extension Functions.

Table 19: Functions in the libslax xutil Extension Library

Function and Arguments	Description	First Supported Junos OS Release
`xutil:json-to-xml(`*string*`)`	Convert a string containing JSON data into the native representation of that data in XML.	14.2
`xutil:max-call-depth(`*number*`)`	Limit the depth of recursive calls. The default limit is 3000.	14.1
`xutil:string-to-xml(`*string+*`)`	Convert a string containing XML data into the native representation of that data.	12.2
`xutil:xml-to-json(`*node-set+*`)`	Convert XML data into a string containing JSON data. This function encodes tags as JSON objects inside a string.	14.2
`xutil:xml-to-string(`*node-set+*`)`	Convert XML data into a string.	12.2

RELATED DOCUMENTATION

libslax Library and Extension Libraries Overview | 221
libslax Distribution Overview | 220
Download and Install the libslax Distribution | 223
Understanding the SLAX Processor (slaxproc) | 247
How to Use the SLAX Processor (slaxproc) | 254
SLAX Debugger, Profiler, and callflow | 260

Understanding the SLAX Processor (slaxproc)

IN THIS SECTION

slaxproc Overview

The libslax distribution contains the libslax library, which incorporates a SLAX writer and SLAX parser, a debugger, a profiler, and the SLAX processor (slaxproc). The SLAX processor is a command-line tool that can validate SLAX script syntax, convert between SLAX and XSLT formats, and format, debug, or run SLAX scripts.

The SLAX processor is invoked on the command line using the `slaxproc` command. The `slaxproc` command accepts command-line arguments that specify the mode of the processor, any behavioral options, and required input and output files.

The syntax for the `slaxproc` command is:

```
slaxproc [mode] [options] [script] [files]
```

The slaxproc mode defines what function the processor performs. slaxproc options include file options and common options. File options are used to specify the script file, input file, output file, and trace file. Common options include additional functionality provided by the SLAX processor such as verbose debugging output.

You can access the slaxproc help by issuing the `slaxproc` command with the `--help` or `-h` option.

```
$ slaxproc -h
```

For more information about the SLAX processor, see slaxproc: The SLAX Processor.

slaxproc Modes

The slaxproc mode defines what function the processor performs. The default mode is `--run` or `-r`. If you do not explicitly specify a mode, the SLAX processor executes a script. Table 20 on page 248 outlines the slaxproc modes and their functions.

Table 20: Slaxproc Modes

Mode	Description
`--check` `-c`	Perform a syntax and content check on a SLAX script, reporting any errors. This mode is useful for off-box syntax checks before installing or uploading scripts to a device running Junos OS.
`--format` `-F`	Format a SLAX script, correcting indentation and spacing to the preferred style.
`--json-to-xml`	Transform JSON input into XML using the conventions defined in http://juniper.github.io/libslax/slax-manual.html#json-elements.
`--run` `-r`	Run a SLAX script. This is the default mode. The script name, input filename, and output filename can be provided using command-line options, positional arguments, or a mix of both. Input defaults to standard input, and output defaults to standard output.
`--show-select`	Show an XPath selection from the input document. This mode is used to extract out selections from a script for external consumption. This enables the consumer to avoid a SLAX parser but still have visibility into the contents of the script.
`--show-variable`	Show the contents of a global variable. This mode is used to extract static variable contents for external consumption. This enables the consumer of the data to avoid a SLAX parser but still have access to the static contents of global variables, such as the `$arguments` variable.
`--slax-to-xslt` `-x`	Convert a SLAX script into XSLT format. The script filename and output filename are provided using command-line options, positional arguments, or a mix of both.
`--xml-to-json`	Convert XML into JSON using the conventions defined in http://juniper.github.io/libslax/slax-manual.html#json-elements.

Table 20: Slaxproc Modes *(Continued)*

Mode	Description
`--xpath` *`xpath`* `-X` *`xpath`*	Select data from the input document that matches the given XPath expression.
`--xslt-to-slax` `-s`	Convert an XSLT script into SLAX format. The script filename and output filename are provided using command-line options, positional arguments, or a mix of both.

slaxproc Options

The slaxproc options include file options and common options. File options are used specify the script file, input file, output file, and trace file. Common options include additional functionality and options provided by the SLAX processor such as verbose debugging output. Table 21 on page 249 lists the slaxproc common options and file options.

Table 21: slaxproc Common Options and File Options

Option	Description
`--debug` `-d`	Enable the SLAX/XSLT debugger.
`--empty` `-E`	Provide an empty document as the input data set. This is useful for scripts that do not expect or need meaningful input.
`--expression <expr>`	Convert a SLAX expression to an XPATH expression, or vice versa, depending on the presence of --slax-to-xslt or --xslt-to-slax.
`--exslt` `-e`	Enable the EXSLT library, which provides a set of standard extension functions. See http://www.exslt.org for more information.
`--help` `-h`	Display the help message and exit.

Table 21: slaxproc Common Options and File Options *(Continued)*

Option	Description
`--html` `-H`	Parse input data using the HTML parser, which differs from XML.
`--ignore-arguments`	Do not process any further arguments. This can be combined with `#!` to enable distinct styles of argument parsing.
`--include <dir>` `-I <dir>`	Add a directory to the list of directories searched when using include and import files. Alternatively, you can define the SLAXPATH environment variable to specify a colon-delimited list of directories to search.
`--indent` `-g`	Indent output. This option is identical to the behavior triggered by `output-method { indent 'true'; }`.
`--input <file>` `-i <file>`	Read input from the specified file.
`--json-tagging`	Tag JSON elements with the 'json' attribute as the input is converted to XML. This enables you to use the `--format` mode to transform the data back into JSON format.
`--keep-text`	When building a script from mini templates, do not add a template to discard normal text. By default, XSLT displays unmatched text data. This option preserves the default behavior instead of replacing it with the discard action.
`--lib <dir>` `-L <dir>`	Adds a directory to the list of directories searched when using extension libraries. Alternatively, you can define the SLAXEXTPATH environment variable to specify a colon-delimited list of extension library locations to search.
`--log <file>`	Write log data to the given file.
`--mini-template <code>` `-m <code>`	Pass a simple script in through the command line. The argument is typically a template, such as a named or match template.

Table 21: slaxproc Common Options and File Options *(Continued)*

Option	Description
--name <file> -n <file>	Read the SLAX script from the specified file.
--no-json-types <dir>	Omit type attributes when using slaxproc mode --json-to-xml to transform JSON input into XML.
--no-randomize	Do not initialize the random number generator. This is useful if you want the script to return identical data for a series of invocations. This option is typically only used during testing.
--no-tty	Do not use tty for the SLAX debugger and other tty-related input needs.
--output <file> -o <file>	Write output to the specified file.
--param <name> <value> -a <name> <value>	Pass a parameter to the script using the name and value pair provided. All parameters are string parameters, so normal quoting rules apply.
--partial -p	Allow the input data to contain a partial SLAX script, which can be used with the --slax-to-xslt or -x mode to perform partial transformations.
--slax-output -S	Write the results using SLAX-style XML.
--trace <file> -t <file>	Write trace data to the specified file.
--verbose -v	Add verbose internal debugging output to the trace data output, including calls to the slaxLog() function.

Table 21: slaxproc Common Options and File Options *(Continued)*

Option	Description
`--version` `-V`	Show version information and exit.
`--write-version <version>` `-w <version>`	Write the specified version number to the output file when converting a script using the `--xslt-to-slax` or `-s` mode. This option can be used to limit the conversion to only use features in certain versions of SLAX. Acceptable values are 1.0, 1.1, 1.2. If this option is not specified, the SLAX script version defaults to the latest version.

slaxproc File Argument Handling

For all modes except `check`, you have the option to reference file arguments positionally or use the file options to specify input and output files. If you use the file options, the files can be referenced in any order on the command line, and the file options can be interspersed among other command-line options.

If no input file is required, use the `-E` option to indicate an empty input document. Additionally, if the input or output option argument has the value "-", the standard input or standard output file is used. When using standard input, press Ctrl+d to signal the end-of-file.

To reference files positionally on the command line, specify the script file first if it is required for that mode, then specify the input file, and lastly specify the output file. Referencing the files positionally allows slaxproc to be plug compatible with xsltproc.

```
$ slaxproc script.slax input.xml output.xml
```

To reference files using explicit file option values, include `--name` or `-n`, `--input` or `-i`, and `--output` or `-o`, to specify the SLAX script file, and the input and output files, respectively.

```
$ slaxproc -i input.xml -n script.slax -o output.xml
```

If a file option is not provided, the filename is parsed positionally. In the following command, the input and output filenames are specified using the file options, but the script filename is referenced positionally:

```
$ slaxproc -i input.xml -o output.xml -g -v script.slax
```

To execute a script that requires no input file, include the `-E` option to indicate an empty input document.

```
$ slaxproc -E script.slax output.xml
```

slaxproc UNIX Scripting Support

SLAX supports the shebang construct (#!), allowing the first line of a script to begin with the characters "#" and "!" followed by a path to the executable that runs the script and a set of command-line arguments. For example:

```
#!/usr/bin/slaxproc -n

or

#!/opt/local/bin/slaxproc -n
```

The operating system adds the name of the scripts and any command-line arguments to the command line that follows the "#!". Adding the `-n` option allows additional arguments to be passed in on the command line. Flexible argument parsing allows aliases. For example, if the first line of the script is:

```
#!/usr/bin/slaxproc -E -n
```

additional arguments can be provided:

```
$ that-script -g output.xml
```

and the resulting command becomes:

```
/usr/bin/slaxproc -E -n /path/to/that-script -g output.xml
```

If the input or output argument has the value "-", the standard input or standard output file is used. This enables slaxproc to be used as a traditional UNIX filter.

RELATED DOCUMENTATION

How to Use the SLAX Processor (slaxproc)

IN THIS SECTION

The SLAX processor (slaxproc) is a command-line tool that can validate SLAX script syntax, convert between SLAX and XSLT formats, and format or run SLAX scripts. The slaxproc modes define what function the processor performs. The following sections outlines commonly used modes. For a complete list of the slaxproc modes, see "Understanding the SLAX Processor (slaxproc)" on page 247.

Validating SLAX Script Syntax

The SLAX processor provides an option to check the syntax of a SLAX script.

- To check the syntax of a SLAX script, issue the `slaxproc` command with the `--check` or `-c` mode option and the script filename.

```
$ slaxproc --check script1.slax

  OR
```

```
$ slaxproc -c script1.slax
```

If the script syntax is correct, the SLAX processor issues a "script check succeeds" message. Otherwise, the processor issues a list of error messages detected during script parsing. Fix any indicated errors, and repeat the check.

Converting Scripts Between XSLT and SLAX Formats

The SLAX processor supports converting scripts between SLAX and XSLT formats. When you convert a script, you have the option to reference the file arguments positionally or use the command-line file options, `--input` or `-i` and `--output` or `-o`, to specify the original input script and the converted output script, respectively. If you use the command-line file options, the files can be referenced in any order on the command line, and the file options can be interspersed among other command-line options.

If you do not provide an argument specifying an input file or an output file, standard input or standard output is used. When using standard input, press Ctrl+d to signal the end-of-file.

To convert a SLAX script to XSLT, issue the `slaxproc` command with the `--slax-to-xslt` or `-x` mode option. To reference the files positionally, specify the input SLAX file as the first argument and the desired output path and filename of the converted XSLT script as the second argument. To reference the files using command-line file options, include the file options in any order. For example:

```
$ slaxproc --slax-to-xslt  test/script2.slax  test/script2.xsl

  OR

$ slaxproc -x  -i test/script2.slax  -o test/script2.xsl
```

To convert an XSLT script to SLAX, issue the `slaxproc` command with the `--xslt-to-slax` or `-s` mode option. To reference the files positionally, specify the input XSLT file as the first argument and the desired output path and filename of the converted SLAX script as the second argument. To reference the files using command-line file options, include the file options in any order.

Optionally, when converting a script from XSLT to SLAX, include the `--write-version` or `-w` option to specify the SLAX version of the converted script. Acceptable values are 1.0, 1.1, and 1.2. The default version is the latest SLAX version. Use the `-p` option for partial input when you do not require the SLAX script boilerplate in the output.

The following example converts the XSLT script **script1.xsl** to the SLAX script **script1.slax**. The SLAX script will include the statement "version 1.0;" as the first line of the script.

```
$ slaxproc --xslt-to-slax -w 1.0 test/script1.xsl  test/script1.slax

  OR

$ slaxproc -s -w 1.0 -i test/script1.xsl -o test/script1.slax
```

The slaxproc `--xslt-to-slax` mode with the `-p` option is useful for quickly converting Junos OS hierarchies from XML format into SLAX. The following example provides the Junos OS `[edit policy-options]` hierarchy in XML format as input to the SLAX processor. The `-p` option indicates partial script input as opposed to a full script.

```
$ slaxproc -s -p
<policy-options>
    <policy-statement>
        <name>export-policy</name>
        <term>
            <name>term1</name>
            <from>
                <route-filter>
                    <address>10.0.4.4/30</address>
                    <prefix-length-range>/30-/30</prefix-length-range>
                </route-filter>
            </from>
            <then>
                <accept/>
            </then>
        </term>
    </policy-statement>
</policy-options>
[Ctrl+d]
```

The SLAX processor returns the SLAX formatting for the hierarchy.

```
<policy-options> {
    <policy-statement> {
        <name> "export-policy";
        <term> {
            <name> "term1";
```

```
            <from> {
                <route-filter> {
                    <address> "10.0.4.4/30";
                    <prefix-length-range> "/30-/30";
                }
            }
            <then> {
                <accept>;
            }
        }
    }
}
```

Running SLAX Scripts

The SLAX processor supports executing SLAX scripts from the command line. This is the default slaxproc mode. To explicitly use this mode, issue the `slaxproc` command with the `--run` or `-r` command-line mode option.

When you execute a script, you have the option to reference the file arguments positionally or use the command-line file options, `--name` or `-n`, `--input` or `-i`, and `--output` or `-o`, to specify the SLAX script file, and the input and output files, respectively. If you use the command-line file options, the files can be referenced in any order on the command line, and the file options can be interspersed among other command-line options.

If no input file is required, use the `-E` option to indicate an empty input document. Additionally, if the input or output argument has the value "-", standard input or standard output is used. When using standard input, press Ctrl+d to signal the end-of-file.

The syntax for executing a script is:

```
$ slaxproc script input-file output-file

OR

$ slaxproc (--name | -n) script (--input | -i) input-file (--output | -o) output-file
```

To execute a script using the slaxproc command-line tool:

1. Create a script using your favorite editor.

2. (Optional) Check the script syntax by invoking the processor with the `--check` or `-c` mode option, and fix any indicated errors.

```
$ slaxproc -c test/script1.slax
```

3. Execute the script and provide the required input and output files as well as any desired slaxproc options.

   You can reference files positionally or use the command-line file options.

   - To execute a script named **script1.slax** using **input.xml** as the input document and **output.xml** as the output document, issue either of the following commands. The two commands are identical in execution.

   ```
   $ slaxproc script1.slax input.xml output.xml

   $ slaxproc -n script1.slax -i input.xml -o output.xml
   ```

   - To execute a script that requires no input file, include the `-E` option to indicate an empty input document. For example:

   ```
   $ slaxproc -E script1.slax output.xml

   $ slaxproc -n script1.slax -o output.xml -E
   ```

   - To execute a script and use standard input as the input document, issue the `slaxproc` command with no input file argument. At the prompt, enter the input and press Ctrl+d to signal the end-of-file. For example:

   ```
   $ slaxproc -n script1.slax -o output.xml
   <user input>
   [Ctrl+d]
   ```

Formatting SLAX Scripts

The SLAX processor provides the option to format a script to correct the indentation and spacing to the preferred style. When you format a script, you have the option to reference the file arguments positionally or use the command-line file options, `--input` or `-i` and `--output` or `-o`, to specify the unformatted input file and the formatted output file, respectively. If you use the command-line file options, the files can be referenced in any order on the command line.

To format a SLAX script, issue the `slaxproc` command with the `--format` or `-F` mode option. To reference the files positionally, specify the unformatted SLAX script as the first argument and the desired output path and filename of the formatted SLAX script as the second argument. To reference the files using command-line file options, include the file options in any order. For example:

```
$ slaxproc --format  script1.slax  script1-format.slax

  OR

$ slaxproc -F  -i script1.slax  -o script1-format.slax
```

Given the following unformatted SLAX script as input:

```
version 1.1;

decimal-format default-format {
decimal-separator "." ;
digit "#" ;
grouping-separator "," ;
infinity "Infinity" ;
minus-sign "-" ;
nan "NaN";
pattern-separator ";" ;
percent "%";
per-mille "\x2030";
zero-digit "0" ;
}

match / {
var $number = -14560302.5;
expr format-number($number, "###,###.00", "default-format");
}
```

the SLAX processor outputs the following formatted SLAX script:

```
version 1.1;

decimal-format default-format {
    decimal-separator ".";
    digit "#";
    grouping-separator ",";
```

```
    infinity "Infinity";
    minus-sign "-";
    pattern-separator ";";
    percent "%";
    per-mille " 30";
    zero-digit "0";
    nan "NaN";
}

match / {
    var $number = -14560302.5;

    expr format-number($number, "###,###.00", "default-format");
}
```

RELATED DOCUMENTATION

libslax Distribution Overview | 220

libslax Library and Extension Libraries Overview | 221

Understanding the SLAX Processor (slaxproc) | 247

SLAX Debugger, Profiler, and callflow

IN THIS SECTION

SLAX Debugger, Profiler, and callflow Overview

The Junos OS command-line interface (CLI) and the libslax distribution include the SLAX debugger (sdb), which is used to trace the execution of SLAX scripts. The SLAX debugger enables you to step through script execution, pause script execution at defined breakpoints, and review the value of script variables at any point.

The SLAX debugger operation and command syntax resemble that of the GNU Project Debugger (GDB). Many of the sdb commands follow their GDB counterparts, to the extent possible. Table 22 on page 261 lists the SLAX debugger commands and a brief description of each command.

The SLAX debugger includes a profiler that can report information about the activity and performance of a script. The profiler, which is automatically enabled when you start the debugger, tracks script execution until the script terminates. At any point, profiling information can be displayed or cleared, and the profiler can be temporarily disabled or enabled. The SLAX debugger `callflow` command enables printing of informational data when you enter or exit levels of the script.

Table 22: SLAX Debugger Commands

Command	Description
break [*loc*]	Add a breakpoint to the script at the current line of execution. Optionally specify [*file*:]*line* or a template name to create a breakpoint at that position.
callflow [on | off]	Enable or disable callflow tracing. You can explicitly specify the `on` or `off` value. Omitting the value toggles callflow on and off.
continue [*loc*]	Continue running the script until it reaches the next breakpoint. If there are no defined breakpoints, the script runs in its entirety. Optionally, specify [*file*:]*line* or a template name. When you include the optional argument, script execution continues until it reaches either a breakpoint or the specified line number or template name, whichever comes first.
delete [*num*]	Delete one or all breakpoints. Breakpoints are numbered sequentially as they are created. Omit the optional argument to delete all breakpoints. Include the breakpoint number as an argument to delete only the specified breakpoint. View currently active breakpoints with the `info` command.
finish	Finish executing the current template.
help	Display the help message.
info [breakpoints | profile | profile brief]	Display information about the current script. The default command lists all breakpoints in the script. Optionally specify the `profile` or `profile brief` arguments to display profiling information.

Table 22: SLAX Debugger Commands *(Continued)*

Command	Description
list [*loc*]	List the contents of the current script. Optionally specify [*file*:]*line* or a template name from which point the debugger lists partial script contents. The output includes the filename, line number, and code.
next	Execute the next instruction, stepping over any function or template calls.
over	Execute the next instruction, stepping over any function or template calls or instruction hierarchies.
print <*xpath*>	Print the value of the XPath expression.
profile [clear | on | off | report | report brief]	Enable or disable the profiler. The profiler is enabled by default. Include the `clear` option to clear profiling information. Include the `report` or `report brief` option to display profiling information for the current script.
quit	Exit debugging mode.
reload	Reload the script.
run	Restart script execution from the beginning of the script.
step	Execute the next instruction, stepping into any function or template calls or instruction hierarchies.
where	Show the backtrace of template calls.

How to Use the SLAX Debugger, Profiler, and callflow

IN THIS SECTION

Invoking the SLAX Debugger

Both the Junos OS CLI and the SLAX processor in the libslax distribution include the SLAX debugger (sdb), which is used to trace the execution of SLAX scripts.

When you invoke the SLAX debugger, the command-line prompt changes to (sdb) to indicate that you are in debugging mode. For example:

```
sdb: The SLAX Debugger (version )
Type 'help' for help
(sdb)
```

When using the SLAX debugger from the Junos OS CLI, you can only use the debugger with op scripts that are enabled in the configuration. To invoke the SLAX debugger from the CLI on a device running Junos OS, issue the `op invoke-debugger cli` operational mode command, include the op script name, and optionally include any necessary script arguments.

```
user@host> op invoke-debugger cli script <argument-name argument-value>
```

The following example invokes the SLAX debugger for the op script **ge-interfaces.slax**, which has two parameters, `interface` and `protocol`. Values are supplied for both arguments.

```
user@host> op invoke-debugger cli ge-interfaces interface ge-0/2/0.0 protocol inet
sdb: The SLAX Debugger (version )
Type 'help' for help
(sdb)
```

To invoke the SLAX debugger when using the SLAX processor, issue the `slaxproc` command with the `--debug` or `-d` option. Specify the script file and any input or output files. If no input file is required, use the `-E` option to indicate an empty input document. If the `-i` or `--input` argument has the value "-", or if you do not include the input option or an input file, standard input is used. When using standard input, press Ctrl+d to signal the end-of-file. The general syntax is:

```
$ slaxproc --debug [options] [script] [files]
```

The following example invokes the SLAX debugger for the script **script1.slax** with an empty input document and an output file **script1-output.xml**

```
$ slaxproc --debug -n script1.slax -o script1-output.xml -E
sdb: The SLAX Debugger (version )
Type 'help' for help
(sdb)
```

Using the SLAX Debugger (sdb)

To view the SLAX debugger help message, issue the `help` command at the (sdb) prompt. To display the help message for a single command, issue `help` *command*, where *command* is the sdb command for which you want more information. For example:

```
(sdb) help break
  break [loc]     Add a breakpoint at [file:]line or template
```

The process for debugging a script varies depending on the script. A generic outline is presented here:

1. Enter debugging mode.
2. Insert breakpoints in the script using the `break` command.

   During execution, the debugger pauses at defined breakpoints.

   The breakpoint location can be the name of a template or a line number in the current script, or the filename and a line number separated by a colon. If you do not include an argument, a breakpoint is created at the current line of execution. Breakpoints are numbered sequentially as you create them. To view a list of breakpoints, issue the `info breakpoints` command. To delete a breakpoint, issue the `delete` *num* command, and specify the breakpoint number. To delete all breakpoints, issue the `delete` command with no argument.

   The following example creates three breakpoints, the first at line 7, the second at line 25, and the third at the template named "three":

   ```
   (sdb) break 7
   Breakpoint 1 at file script1.slax, line 7
   ```

```
(sdb) break 25
Breakpoint 2 at file script1.slax, line 25
(sdb) break three
Breakpoint 3 at file script1.slax, line 51
(sdb) info breakpoints
List of breakpoints:
    #1 [global] at script1.slax:7
    #2 template two at script1.slax:25
    #3 template three at script1.slax:51
```

3. Increment script execution by issuing the `continue`, `finish`, `next`, `over`, and `step` commands at the debugger prompt, for example:

```
(sdb) next
Reached breakpoint 1, at script1.slax:7
script1.slax:3: var $byte = "10011001";
```

4. Review the value of variables as the program executes to ensure that they have the expected value.

```
print xpath-expression
```

5. To reload the script contents at any point and restart script execution from the beginning, issue the `reload` command.

```
(sdb) reload
The script being debugged has been started already.
Reload and restart it from the beginning? (y or n) y
Reloading script...
Reloading complete.
```

Using the SLAX Profiler

The SLAX debugger includes a profiler that can report information about the activity and performance of a script. The profiler, which is automatically enabled when you start the debugger, tracks script execution until the script terminates. At any point, profiling information can be displayed or cleared, and the profiler can be temporarily disabled or enabled.

To access the profiler, issue the profile command at the SLAX debugger prompt, (sdb), and include any options. The profile command syntax is:

```
(sdb) profile [options]
```

Table 23 on page 266 lists the profile command options. Issuing the profile command with no additional options toggles the profiler on and off.

```
(sdb) profile
Disabling profiler
(sdb)
```

You can access the profiler help by issuing the help profile command at the (sdb) prompt.

Table 23: Profile Command Options

Option	Description
clear	Clear profiling information
off	Disable profiling
on	Enable profiling
report [brief]	Report profiling information

To enable the profiler and print a report:

1. Enter debugging mode. The profiler is enabled by default.
2. Step through script execution, or execute a script in its entirety.

```
(sdb) run
<?xml version="1.0"?>
<message>Down rev PIC in Fruvenator, Fru-Master 3000</message>
Script exited normally.
```

3. At any point during script execution, display profiling information.

The `brief` option instructs sdb to avoid showing lines that were not hit, since there is no valid information. If you omit the `brief` option, dashes are displayed.

```
(sdb) profile report brief
```

The following sample output shows a profile report with and without the `brief` option. The source code data in the example is truncated for display purposes.

```
(sdb) profile report
 Line   Hits   User    U/Hit  System    S/Hit Source
    1      -      -        -       -        - version 1.0;
    2      -      -        -       -        -
    3      2      4     2.00       8     4.00 match / {
    4      1     25    25.00      13    13.00     var ....
    5      -      -        -       -        -
    6      -      -        -       -        -     for-each....
    7      1     45    45.00      10    10.00          ..
    8      1     12    12.00       5     5.00         <message>
    9      1     45    45.00      15    15.00          ....
   10      -      -        -       -        -     }
   11      -      -        -       -        - }
Total      6    131               51    Total

(sdb) pro rep b
 Line   Hits   User    U/Hit  System    S/Hit Source
    3      2      4     2.00       8     4.00 match / {
    4      1     25    25.00      13    13.00     var  ....
    7      1     45    45.00      10    10.00          ....
    8      1     12    12.00       5     5.00      <message>
    9      1     45    45.00      15    15.00          ....
Total      6    131               51    Total
```

The profile report includes the following information:

- `Line`—Line number in the source file.
- `Hits`—Number of times this line was executed.
- `User`—Number of microseconds of "user" time spent processing this line.
- `U/Hit`—Average number of microseconds of "user" time per hit.
- `System`—Number of microseconds of "system" time spent processing this line.

- `S/Hit`—Average number of microseconds of "system" time per hit.
- `Source`—Source code line.

This information not only shows how much time is spent during code execution, but can also show which lines are being executed, which can help debug scripts where the execution does not match expectations.

Using callflow

The SLAX debugger `callflow` command enables printing of informational data when you enter or exit levels of the script.

To enable callflow and view callflow data for a script:

1. Enter debugging mode.
2. Issue the `callflow` command at the SLAX debugger prompt, (sdb).

```
(sdb) callflow
Enabling callflow
```

3. Step through script execution, or execute a script in its entirety.

   callflow prints information as it enters and exits different levels of the script. Each output line references the instruction, filename, and line number of the frame.

```
(sdb) run
callflow: 0: enter <xsl:template> in match / at script3.slax:5
callflow: 1: enter <xsl:template> in template one at script3.slax:14
callflow: 2: enter <xsl:template> in template two at script3.slax:20
callflow: 3: enter <xsl:call-template> at script3.slax:22
...
<?xml version="1.0"?>
<message>Down rev PIC in Fruvenator, Fru-Master 3000</message>
Script exited normally.
```

RELATED DOCUMENTATION

libslax Distribution Overview | 220
libslax Library and Extension Libraries Overview | 221
Understanding the SLAX Processor (slaxproc) | 247

How to Use the SLAX Processor (slaxproc) | 254

RELATED DOCUMENTATION

op invoke-debugger cli
Understanding the SLAX Processor (slaxproc) | 247
How to Use the SLAX Processor (slaxproc) | 254
libslax Distribution Overview | 220

4

PART

Automation Scripting Using Python

CHAPTER 9

Python Overview

IN THIS CHAPTER

Understanding Python Automation Scripts for Junos Devices

IN THIS SECTION

Python Scripts Overview

As an alternative to SLAX and XSLT, you can create and execute Python scripts on Junos devices that support the Python extensions package in the software image. Python is a practical, feature-rich language with extensive library support that enables you to create custom scripts. Table 24 on page 272 outlines the Python versions available on Junos devices and notes the release in which support for that version was added or removed for certain types of scripts.

Table 24: Python Versions on Devices Running Junos OS

Python version	Release	Change	Affected Script Types
Python 2.7.x	Junos OS Release 16.1R1 and later	Support added	Commit, event, op, and SNMP scripts Juniper Extension Toolkit (JET) scripts YANG action and translation scripts
	Junos OS Release 20.2R1 and later	Support removed	YANG action and translation scripts
	Junos OS Release 21.1R1 and later	Support removed	Commit, event, op, and SNMP scripts Juniper Extension Toolkit (JET) scripts
	Junos OS Evolved Release 22.3R1 and later	Support removed	Commit, event, op, and SNMP scripts Juniper Extension Toolkit (JET) scripts YANG action and translation scripts
Python 3.x	Junos OS Release 19.4R1 and later	Support added	Commit, event, op, and SNMP scripts
	Junos OS Release 20.2R1 and later	Support added	Juniper Extension Toolkit (JET) scripts YANG action and translation scripts
	Junos OS Evolved Release 21.1R1 and later	Support added See Feature Explorer for supported platforms.	Commit, event, op, and SNMP scripts Juniper Extension Toolkit (JET) scripts YANG action and translation scripts

NOTE: The Python 2.7 end-of-life and end-of-support date was January 1, 2020. The official upgrade path for Python 2.7 is to Python 3. We recommend that you migrate supported script types from Python 2 to Python 3.

Python scripts can leverage all of the features and constructs in the Python language, which provides flexibility over SLAX and XSLT and enables you to perform operations that are difficult or impossible to perform in SLAX and XSLT. SLAX and XSLT are designed for processing XML data, but Python is suited for processing any format. Python supports an extensive list of data types, including lists, dictionaries, sets, and tuples, and many Python variables are mutable, unlike many SLAX and XSLT variables. SLAX and XSLT scripts have limited loop control, but Python scripts can utilize statements like `break` and `continue` to precisely control loop behavior. Python also has extensive support for file operations, and you can invoke the standard Python debugger on the command line to debug Python op scripts..

In addition, Junos OS supports the following in Python scripts:

- Junos PyEZ library—simplifies executing RPCs and performing operational and configuration tasks on devices running Junos OS.
- `lxml` library—simplifies XPath handling.
- Jinja2 library—template engine that enables you to generate content from predefined templates, which can be useful for generating Junos OS configuration data.
- IPv6—Starting in Junos OS Release 19.3R1, devices running Junos OS with upgraded FreeBSD support using IPv6 in Python automation scripts. For more information see "IPv6 Support in Python Automation Scripts" on page 308.
- Non-default routing instances—Starting in Junos OS Release 19.3R1, Python scripts on supported devices can use the "set_routing_instance()" on page 412 extension function to connect to a remote device through a nondefault routing instance.
- Additional Python modules—For information about the modules that are available to Python scripts on devices running Junos OS, see "Overview of Python Modules on Junos Devices" on page 280.

To prevent the execution of unauthorized Python code on devices running Junos OS, unsigned Python scripts must meet certain requirements before you can execute the script on a device. For detailed information about the requirements for executing Python automation scripts on devices running Junos OS, see "Requirements for Executing Python Automation Scripts on Junos Devices" on page 276.

To execute unsigned Python scripts using either Python 2.7 or Python 3, you must configure the `language python` or `language python3` statement, respectively, at the `[edit system scripts]` hierarchy level. If you configure the `language python3` statement, the device uses Python 3 to execute scripts that support this Python version, and it uses Python 2.7 to execute scripts that do not support Python 3 in the given Junos OS release, as noted in Table 24 on page 272.

For JET scripts, you must configure the version of the statement that corresponds to the Python version the script supports. For example, if you configure the `language python3` statement for a JET script that only supports Python 2.7, you will receive a runtime error when you execute the script.

Python Script Inputs for Commit, Event, Op, and SNMP Scripts

Devices running Junos OS that support Python automation scripts include the following modules, which you can import into commit, event, op, and SNMP scripts:

- `jcs` module—provides access to a subset of the extension functions and templates that are available to SLAX and XSLT scripts.

  For information about Junos OS extension functions and templates, see "Understanding Extension Functions in Junos OS Automation Scripts" on page 337 and "Understanding Named Templates in Junos OS Automation Scripts" on page 427.

- `junos` module—contains the `Junos` class and script inputs outlined in Table 25 on page 274.

Table 25: junos Module Objects

Python Object	Type	Script Type	Description	XSLT/SLAX Equivalent
`Junos_Context`	Dictionary	Commit Event Op	Device-specific information about the script execution environment such as the device hostname, the script type, and the user executing the script.	commit-script-input/junos-context event-script-input/junos-context op-script-input/junos-context
`Junos_Configuration`	lxml.etree _Element	Commit	Post-inheritance candidate configuration in Junos XML format.	commit-script-input/configuration
`Junos_Trigger_Event`	lxml.etree _Element	Event	Details of the event that triggered the corresponding event policy.	event-script-input/trigger-event
`Junos_Received_Events`	lxml.etree _Element	Event	Details of correlated events that occurred before the trigger event.	event-script-input/received-events

Table 25: junos Module Objects *(Continued)*

Python Object	Type	Script Type	Description	XSLT/SLAX Equivalent
`Junos_Remote_Execution_Details`	Generator function that produces a sequence of remote devices.	Event	Connection details for a local or remote host configured for an event script at the `[edit event-options event-script file` *`filename`* `remote-execution]` hierarchy level.	event-script-input/ remote-execution-details

For detailed information about the different script inputs and how to import and use them in automation scripts, see:

- "Global Parameters and Variables in Junos OS Automation Scripts" on page 327
- "Required Boilerplate for Commit Scripts" on page 542
- "Required Boilerplate for Op Scripts" on page 822
- "Required Boilerplate for Event Scripts" on page 1084
- "Use Event and Remote Execution Details in Event Scripts" on page 1088
- "Required Boilerplate for SNMP Scripts" on page 1124

Python Script Resources

Table 26 on page 275 provides resources to help you create and execute Python scripts on devices running Junos OS. Table 27 on page 276 provides resources for each of the different types of scripts.

Table 26: General Resources

Resource	Documentation
Python script execution requirements	"Requirements for Executing Python Automation Scripts on Junos Devices" on page 276
Python modules on devices running Junos OS	"Overview of Python Modules on Junos Devices" on page 280

Table 26: General Resources *(Continued)*

Resource	Documentation
IPv6 support in Python scripts	"IPv6 Support in Python Automation Scripts" on page 308
Junos PyEZ documentation	Junos PyEZ
Python programming language documentation	https://www.python.org/

Table 27: Resources by Script Type

Script Type	Documentation
Junos OS commit, event, op, and SNMP Scripts	Junos OS Automation Scripting User Guide
Juniper Extension Toolkit (JET) scripts	Juniper Extension Toolkit documentation
YANG action and translation scripts	NETCONF XML Management Protocol Developer Guide

RELATED DOCUMENTATION

language

Requirements for Executing Python Automation Scripts on Junos Devices

IN THIS SECTION

Requirements for Executing Python Scripts

You can use Python to author Junos OS commit, event, op, and SNMP automation scripts. To prevent the execution of unauthorized Python code, Junos devices, by default, do not permit executing unsigned Python scripts. To execute an unsigned Python script, the following requirements must be met, which are in addition to the normal requirements for onbox automation scripts:

- File owner is either root or a user in the Junos OS `super-user` login class.
- Only the file owner has write permission for the file.
- The execution of unsigned Python scripts is enabled by configuring the `language python` or `language python3` statement at the `[edit system scripts]` hierarchy level.

> **NOTE**: If you configure the `language python3` statement, the device uses Python 3 to execute scripts that support this Python version, and it uses Python 2.7 to execute scripts that do not support Python 3 in the given Junos OS release. For more information, see "Understanding Python Automation Scripts for Junos Devices" on page 271.

> **NOTE**: Starting in Junos OS Evolved Release 21.2R1, the `junos-defaults` configuration group includes the `language python` statement by default.

> **NOTE**: To enable a user who does not belong to the file's user or group class to execute an unsigned Python automation script, the script's file permissions must include read permission for others.

As with SLAX and XSLT automation scripts, you must store Python automation scripts in the appropriate directory on the device, and you must enable individual scripts by configuring the script filename under the hierarchy level appropriate to the script type in the configuration. For information about storing and enabling automation scripts, see "Store and Enable Junos Automation Scripts" on page 448.

We recommend that you configure a checksum to verify the integrity of Python scripts. To specify a checksum for a local script, configure the `checksum` statement under the `[file` *`filename`*`]` statement in the hierarchy for your specific type of script. To specify a checksum for a remote op script, include the `key` argument when you execute the script using the `op url` command. Starting in Junos OS Release 18.2R2 and 18.3R1, if you execute an unsigned Python script that does not have a checksum configured, Junos devices log a CSCRIPT_SECURITY_WARNING message in the system log file. For example:

```
CSCRIPT_SECURITY_WARNING: unsigned python script '/var/db/scripts/op/sample.py' without checksum
is executed
```

Configuring Access Privileges for Python Scripts

Starting in Junos OS Release 16.1R3, interactive Python scripts, such as commit and op scripts, run with the access privileges of the user who executes the command or operation that invokes the script. Non-interactive Python scripts, such as event and SNMP scripts, by default, execute under the privileges of the user and group `nobody`. To execute event or SNMP scripts using the access privileges of a specific user, you must configure the `python-script-user` *`username`* statement at the `[edit event-options event-script file` *`filename`*`]` hierarchy level for event scripts or the `[edit system scripts snmp file` *`filename`*`]` hierarchy level for SNMP scripts, and specify a user configured at the `[edit system login]` hierarchy level.

For example:

```
[edit event-options event-script]
file filename {
    python-script-user username;
}
```

```
[edit system scripts snmp]
file filename {
    python-script-user username;
}
```

NOTE: You cannot configure Python event and SNMP scripts to execute with root access privileges.

Summary of Requirement Changes By Release

Starting in Junos OS Release 16.1R3, ownership and access privilege requirements for some unsigned Python scripts are modified. In Junos OS Release 16.1R2 and earlier releases, unsigned Python commit, event, op, and SNMP scripts must be owned by the root user, and Junos OS executes the scripts using the access privileges of the *nix user and group `nobody`, which is the generic, unprivileged system account.

Starting in Junos OS Release 16.1R3, unsigned Python automation scripts must be owned by either the root user or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file. Furthermore, Python automation scripts can be executed with the access privileges of authorized users.

Table 28 on page 279 outlines the requirements for executing unsigned Python automation scripts in the different Veriexec-enabled versions of Junos OS.

Table 28: Python Automation Script Requirements

Requirement/Restriction	Junos OS Release 16.1R2 or Earlier Release	Junos OS Release 16.1R3 or Later Release
File owner	Root user	Root user or a user in the Junos OS `super-user` login class.
File write permissions	Any	File owner only
`language python` or `language python3` statement must be configured at the `[edit system scripts]` hierarchy level **NOTE**: Python 3 is supported in Junos OS Release 19.4R1 and later releases.	Yes	Yes
Script must be enabled in the configuration under the hierarchy appropriate to that script type	Yes	Yes
Access privileges	All Python automation scripts execute with the access privileges of the user and group `nobody`	Python commit and op scripts execute with the access privileges of the user who invokes the script. Python event and SNMP scripts execute with the access privileges of the user configured in the `python-script-user` statement. If the `python-script-user` statement is not configured, the script executes with the access privileges of the user and group `nobody`.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
22.3R1-EVO	Starting in Junos OS Evolved Release 22.3R1, Python 3 is the default and only supported Python version for executing Python scripts.
21.2R1-EVO	Starting in Junos OS Evolved Release 21.2R1, the `language python` statement is configured by default in the `junos-defaults` configuration group.
16.1R3	Starting in Junos OS Release 16.1R3, ownership and access privilege requirements for some unsigned Python scripts are modified.

RELATED DOCUMENTATION

Store and Enable Junos Automation Scripts | 448
Understanding Python Automation Scripts for Junos Devices | 271
python-script-user

Overview of Python Modules on Junos Devices

IN THIS SECTION

The Python interpreter is included as part of the Junos operating system (Junos OS). Junos OS and Junos OS Evolved include many Python modules, packages, and libraries that can be used in Python applications including commit, event, op, and SNMP automation scripts; Juniper Extension Toolkit (JET) applications; and YANG action and translation scripts. Usage of a specific module depends on the permissions of the user who is executing it.

Python Module Upgrades

Table 29 on page 281 outlines the general updates and affected platforms for each set of module upgrades in a specific release. To determine the initial release in which a certain platform supports a specific version of Python (2.7 or 3) for a given type of script, consult the Feature Explorer tool.

Table 29: Python Module Upgrades

Release	Description of Changes	Supported Platforms
Junos OS Release 18.1R1	Module upgrades for Python 2.7	MX240, MX480, MX960, and vMX routers
Junos OS Release 19.2R1	Module upgrades for Python 2.7	MX Series routers and QFX Series switches
Junos OS Release 19.4R1	Module upgrades for Python 2.7 Module additions for Python 3.x (commit, event, op, and SNMP scripts only)	ACX Series, MX Series, and PTX Series routers EX Series and QFX Series switches SRX Series Services Gateways
Junos OS Release 20.2R1	Module additions for Python 3.x	ACX Series, MX Series, and PTX Series routers EX Series and QFX Series switches SRX Series Services Gateways
Junos OS Evolved Release 21.2R1	Module additions for Python 3.x	PTX Series Routers

Installed Python Modules, Packages, and Libraries

Table 30 on page 282 outlines the Python modules, packages, and libraries that are available in Junos OS and Junos OS Evolved for Python 2.7 and Python 3.

Table 30: Python Modules in Junos OS and Junos OS Evolved

Package/Module	Description	Support on Junos OS		Support on Junos OS Evolved	
		2.7	3.x	2.7	3.x
`appdirs`	Determines the path to the appropriate platform-specific directories for application-specific user data.	Y	Y	Y	Y
`asn1crypto`	Library for parsing and serializing Abstract Syntax Notation One (ASN.1) structures.	Y	Y	Y	Y
`attrs`	Python package that enables you to write classes without writing all the boilerplate code.	–	–	–	Y
`Automat`	Python expression of finite-state machines.	–	–	–	Y
`bcrypt`	Library for password hashing and storage.	–	–	–	Y
`cffi`	C Foreign Function Interface for Python, which enables Python applications to interact with C code.	Y	Y	Y	Y
`chardet`	Universal character encoding detector for Python 2 and 3.	Y	Y	Y	Y
`constantly`	Library that provides symbolic constant support.	–	–	–	Y
`Crypto`	Collection of Python modules in the **PyCrypto** package that implement cryptographic algorithms and protocols. The modules provide various secure hash and encryption functions.	Y	Y	Y	Y

Table 30: Python Modules in Junos OS and Junos OS Evolved *(Continued)*

Package/Module	Description	Support on Junos OS		Support on Junos OS Evolved	
		2.7	3.x	2.7	3.x
`cryptography`	Provides cryptographic recipes and primitives in Python.	Y	Y	Y	Y
`concurrent.futures`	Provides an interface for asynchronously executing callables. Python 3 includes this package as part of the standard library.	Y	Y	Y	Y
`ecdsa`	Provides an implementation of ECDSA cryptography, which can be used to create key pairs, sign messages, and verify signatures.	Y	Y	Y	Y
`enum`	`enum34` package, which provides support for enumerations. Python 3 includes this functionality as part of the standard library.	Y	Y	Y	Y
`foolscap`	New version of Twisted's native RPC protocol.	–	–	–	Y
`future`	Python 2 and Python 3 compatibility library.	–	–	–	Y
`google.protobuf`	Provides support for working with protocol buffers (protobuf).	Y	Y	Y	Y
`grpcio`	Python gRPC tools.	Y	Y	Y	Y
`hyperlink`	Pure-Python implementation of immutable URLs.	–	–	–	Y

Table 30: Python Modules in Junos OS and Junos OS Evolved *(Continued)*

Package/Module	Description	Support on Junos OS		Support on Junos OS Evolved	
		2.7	3.x	2.7	3.x
`idna`	Provides support for the Internationalized Domain Names in Applications (IDNA) protocol, as defined in RFC 5891, *Internationalized Domain Names in Applications (IDNA): Protocol*.	Y	Y	Y	Y
`incremental`	Library for versioning Python projects.	–	–	–	Y
`ipaddress`	Provides capabilities to create, manipulate, and operate on IPv4 and IPv6 addresses. Python 3 includes this module as part of the standard library. **NOTE**: Devices running Junos OS with upgraded FreeBSD support using IPv6 in Python scripts starting in Junos OS Release 19.3R1.	Y	Y	Y	Y
`jinja2`	Fast, secure, designer-friendly templating language for Python. For more information about Jinja2, see http://jinja.pocoo.org/docs/dev/.	Y	Y	Y	Y
`jnpr.junos` (Junos PyEZ or junos-eznc)	Microframework for Python that enables you to automate devices running Junos OS. Junos PyEZ is designed to provide the capabilities that a user would have on the Junos OS command-line interface (CLI) in an environment built for automation tasks. For more information, see Junos PyEZ.	Y	Y	Y	Y

Table 30: Python Modules in Junos OS and Junos OS Evolved *(Continued)*

Package/Module	Description	Support on Junos OS		Support on Junos OS Evolved	
		2.7	3.x	2.7	3.x
`jxmlease`	Python module for converting XML to Python data structures and converting Python data structures to XML. For more information, see https://jxmlease.readthedocs.io/.	Y	Y	Y	Y
`lxml`	XML processing library that combines the speed and XML feature completeness of the C libraries libxml2 and libxslt with the simplicity of a native Python API. For more information, see http://lxml.de.	Y	Y	Y	Y
`MarkupSafe`	Provides the ability to escape and format an XML, HTML, or XHTML markup safe string.	Y	Y	Y	Y
`ncclient`	Facilitates client scripting and application development through the NETCONF protocol. For more information about `ncclient`, including documentation for the external APIs, see http://ncclient.grnet.gr/.	Y	Y	Y	Y
`netaddr`	Network address manipulation library that enables processing of Layer 2 and Layer 3 network addresses.	Y	Y	Y	Y
`packaging`	Core utilities for Python packages.	Y	Y	Y	Y
`paho.mqtt`	Serves as a client class that enables applications to connect to a Message Queue Telemetry Transport (MQTT) broker to publish messages and to subscribe to topics and receive published messages.	Y	Y	Y	Y

Table 30: Python Modules in Junos OS and Junos OS Evolved *(Continued)*

Package/Module	Description	Support on Junos OS		Support on Junos OS Evolved	
		2.7	3.x	2.7	3.x
paramiko	SSH2 protocol library that provides the ability to make SSH2 protocol-based connections. This module supports all major ciphers and hash methods for both client and server modes. **NOTE**: paramiko version 2.1.2 is only supported on devices running Junos OS with upgraded FreeBSD.	Y	Y	Y	Y
pkg_resources	Provides APIs for finding and managing Python package and version dependencies and accessing bundled files and resources.	Y	Y	Y	Y
psutil	Library for retrieving information on running processes and system utilization (CPU, memory, disks, and processes). **NOTE**: psutil is only supported on devices running Junos OS Evolved and devices running Junos OS with upgraded FreeBSD. Devices running Junos OS with upgraded FreeBSD only support a subset of psutil functions. For more information, see "How to Use the psutil Module to Retrieve Process and System Information on Devices Running Junos OS" on page 291	Y	Y	Y	Y
pyang	Extensible YANG validator and converter that enables the processing, validation, and conversion of YANG modules. Junos OS adds support for pyang in Python 3 and removes support for pyang in Python 2.7 starting in Junos OS Release 20.2R1.	Y	Y	Y	–
pyasn1	Python implementation of Abstract Syntax Notation One (ASN.1) types and BER/CER/DER codecs (X.208).	Y	Y	Y	Y

Table 30: Python Modules in Junos OS and Junos OS Evolved *(Continued)*

Package/Module	Description	Support on Junos OS		Support on Junos OS Evolved	
		2.7	3.x	2.7	3.x
`pyasn1-modules`	Collection of ASN.1 data structures expressed as Python classes.	–	–	–	Y
`pycparser`	Parser for the C language written in pure Python. The module can be integrated into applications that need to parse C source code.	Y	Y	Y	Y
`PyNaCl`	Python binding to the Networking and Cryptography library (NaCl).	–	–	–	Y
`PyOpenSSL`	Python wrapper module around a subset of the OpenSSL library.	–	–	–	Y
`pyparsing`	Provides an alternative approach to creating and executing simple grammars.	Y	Y	Y	Y
`python-dateutil`	Provides extensions to the datetime module.	–	–	–	Y
`requests`	Library that enables sending HTTP/1.1 requests using Python.	Y	Y	Y	Y
`scp`	Implementation of the SCP protocol for Paramiko that uses Paramiko transport to send and receive files via the SCP protocol.	Y	Y	Y	Y
`serial`	Module in the **pySerial** package that encapsulates the access for serial ports.	Y	Y	Y	Y

Table 30: Python Modules in Junos OS and Junos OS Evolved *(Continued)*

Package/Module	Description	Support on Junos OS		Support on Junos OS Evolved	
		2.7	3.x	2.7	3.x
`service-identity`	Service identify verification for pyOpenSSL and cryptography.	–	–	–	Y
`setuptools`	Library designed to facilitate packaging Python projects.	–	Y	–	Y
`six`	Python 2 and Python 3 compatibility library.	Y	Y	Y	Y
`thrift`	Provides Python bindings for the Apache Thrift framework. `thrift` is only supported for Python 2.7 and only on devices running Junos OS Release 19.3 and earlier.	Y	–	–	–
`transitions`	Object-oriented state machine implementation.	–	–	–	Y
`twisted`	Event-based framework for internet applications.	–	–	–	Y
`urllib3`	HTTP client for Python.	Y	Y	Y	Y
`yaml`	Module in the **PyYAML** package that is used to serialize and deserialize data in YAML format.	Y	Y	Y	Y
`yamlordereddictloader`	YAML loader and dumper for PyYAML that uses `OrderedDict` objects to retain the mapping order when loading or dumping a file.	–	–	–	Y
`zope.interface`	Package that enables you to specify interfaces for Python.	–	–	–	Y

Additional Python Libraries

Certain devices running Junos OS support a limited version of the Scapy Python library. Scapy is an interactive packet manipulation library that enables you to create, send, receive, and dissect network packets. Junos devices support a limited set of modified Scapy APIs.

On Junos OS, you can use Scapy to create, send, and receive IP packets, TCP segments, or UDP datagrams. Thus, Scapy can help gather information or troubleshoot issues in your network. For example, you can calculate the round-trip time to a specific host

Table 31 on page 289 provides links to various Scapy resources. To use Scapy on supported Junos devices, you must install it as a JET application. See the Juniper README for installation options and instructions, information about supported Scapy functions, and usage examples.

Table 31: Scapy Resources

Resource	URL
Scapy website	https://scapy.net
Scapy official documentation	https://scapy.readthedocs.io/en/latest
Scapy GitHub repository	https://github.com/secdev/scapy
Juniper Scapy GitHub repository and documentation	https://github.com/Juniper/scapy https://github.com/Juniper/scapy/blob/master/README.md

RELATED DOCUMENTATION

Understanding Python Automation Scripts for Junos Devices | 271

How to Use Python Interactive Mode on Devices Running Junos OS

Different variants of Junos OS have different restrictions with respect to executing Python scripts on the device. The Junos OS with Enhanced Automation software image is a full-featured version of Junos OS that bundles additional automation tools with the image and disables Veriexec enforcement. On Junos OS variants that have Veriexec enforcement enabled, unsigned Python scripts must meet certain requirements before you can execute them on the device, and you can only execute the scripts using Python's normal script mode in which scripts are run in the Python interpreter. By contrast, devices running Junos OS with Enhanced Automation, which are generally used in development environments, enable you to run unsigned Python scripts with fewer restrictions and also use Python interactive mode in the shell.

In addition, the Junos OS Evolved image runs natively on Linux, providing direct access to all the Linux utilities and operations, including Python and the Python libraries that are part of the base image. Thus devices running Junos OS Evolved enable you to run Python scripts and use Python in interactive mode in the Linux shell.

To invoke Python 2.7 or Python 3 in interactive mode in supported releases on Junos devices, enter the `python` or `python3` command, respectively, at the shell prompt.

To start Python in interactive mode:

1. Start the shell interface:

```
user@host> start shell
```

2. Enter the `python` or `python3` command without any parameters:

```
[vrf:none] user@host:~$ python3
Python 3.5.2 (default, Aug  7 2020, 16:43:19)
[GCC 6.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

NOTE: The Python interpreter is designated with the prompt >>> at the beginning of a line or ... to indicate the continuation of a line.

RELATED DOCUMENTATION

Overview of Junos Automation Enhancements on Devices Running Junos OS with Enhanced Automation | 5

Installing Junos OS Software with Junos Automation Enhancements

FAQ: Junos Automation Enhancements

How to Use the psutil Module to Retrieve Process and System Information on Devices Running Junos OS

The psutil Python module is available on certain devices that support Python automation scripts and that are running either Junos OS Evolved or Junos OS with upgraded FreeBSD. You can use the psutil module in Python scripts to retrieve information about running processes and system utilization on the device, for example, information about the CPU, memory, disks, and processes. The module implements the functionality of many command-line tools such as ps and uptime, among others.

Table 32 on page 291 outlines the supported psutil functions. For more information about the psutil module and its functions, see the official documentation at https://psutil.readthedocs.io/en/latest/.

Table 32: psutil Module Functions

Function Category	Supported Functions (Junos OS with Upgraded FreeBSD)	Supported Functions (Junos OS Evolved)
CPU	cpu_count()	cpu_count() cpu_freq() cpu_percent() cpu_stats() cpu_times()cpu_times_percent()
Disk	disk_partitions() disk_usage()	disk_io_counters() disk_partitions() disk_usage()
Memory	virtual_memory()	swap_memory() virtual_memory()
Network	–	net_connections() net_if_addrs() net_if_stats() net_io_counters()

Table 32: psutil Module Functions *(Continued)*

Function Category	Supported Functions (Junos OS with Upgraded FreeBSD)	Supported Functions (Junos OS Evolved)
Processes	pid_exists() pids() process_iter() wait_procs()	pid_exists() pids() process_iter() wait_procs()
Sensors	–	–
System Info	boot_time() users()	boot_time() users()

The following sample Python op script demonstrates calls to the `psutil` functions to retrieve information about the system and processes on the given device running Junos OS:

```
import psutil
import datetime

### *** CPU FUNCTIONS ***

# Number of logical CPUs in the system
print ("psutil.cpu_count() = {0}".format(psutil.cpu_count()))


### *** DISK FUNCTIONS ***

# List of named tuples containing all mounted disk partitions
dparts = psutil.disk_partitions()
print("psutil.disk_partitions() = {0}".format(dparts))

# Disk usage statistics
du = psutil.disk_usage('/')
print("psutil.disk_usage('/') = {0}".format(du))


### *** MEMORY FUNCTIONS ***
```

```
# System memory usage statistics
mem = psutil.virtual_memory()
print("psutil.virtual_memory() = {0}".format(mem))

THRESHOLD = 100 * 1024 * 1024  # 100MB
if mem.available <= THRESHOLD:
    print("warning, available memory below threshold")


### *** PROCESS FUNCTIONS ***

# List of current running process IDs.
pids = psutil.pids()
print("psutil.pids() = {0}".format(pids))


# Check whether the given PID exists in the current process list.
for proc in psutil.process_iter():
    try:
        pinfo = proc.as_dict(attrs=['pid', 'name'])
    except psutil.NoSuchProcess:
        pass
    else:
        print(pinfo)


### *** SYSTEM INFORMATION FUNCTIONS ***

# System boot time expressed in seconds since the epoch
boot_time = psutil.boot_time()
print("psutil.boot_time() = {0}".format(boot_time))

# System boot time converted to human readable format
print(datetime.datetime.fromtimestamp(psutil.boot_time()).strftime("%Y-%m-%d %H:%M:%S"))

# Users currently connected on the system
users = psutil.users()
print("psutil.users() = {0}".format(users))
```

When you execute the op script, the script prints the requested information about the device. Some of the sample output has been truncated for brevity.

```
user@host> op psutil-test.py
psutil.cpu_count() = 4
psutil.disk_partitions() = [sdiskpart(device='/dev/md0.uzip', mountpoint='/', fstype='cd9660',
opts='ro'), sdiskpart(device='devfs', mountpoint='/dev', fstype='devfs',
opts='rw,multilabel'), ...]
psutil.disk_usage('/') = sdiskusage(total=20609945600L, used=12531843072L, free=6429306880L,
percent=66.099999999999994)
psutil.virtual_memory() = svmem(total=4230012928L, available=7632039936L,
percent=-80.400000000000006, used=658825216L, free=4325273600L, active=59793408L,
inactive=3306766336L, buffers=289771520L, cached=0L, shared=249659392L, wired=599031808L)
psutil.pids() = [43521, 43134, 33616, 33610, 33609, 33608, 33605, 33604, 33603, 33602, 33599,
33598, 33597, 33596, 33593, 8356, 7893, 7871, 7870, 7869, 7868, 7867, 7866, 7865, 7864, 7863,
7862, 7861, 7860, 7859, 7858, 7857, 7856, 7854, 7853, 7851, 7850, 7849, 7848, 7847, 7846, 7845,
7844, 7842, 7841, 7840, 7839, 7838, 7837, 7836, 7835, 7834, 7833, 7832, 7831, 7830, 7829, 7828,
7826, 7825, 7824, 7823, 7822, 7821, 7820, 7819, 7817, 7807, 7627, 7560, 7410, 7370, 7362, 7359,
7345, 7344, 7343, 7342, 7340, 7336, 7335, 7328, 7327, 7322, 7320, 7319, 7318, 7314, 7313, 7312,
7308, 7307, 7304, 7303, 7301, 7299, 7296, 7295, 7293, 7282, 7267, 7266, 7262, 6278, 6276, 6275,
5886, 5493, 5492, 4015, 4014, 3954, 3953, 3895, 3894, 3835, 3834, 3776, 3775, 3717, 3716, 3660,
3659, 3601, 3600, 3541, 3540, 3481, 3480, 3423, 3422, 3364, 3363, 3304, 3303, 3160, 3159, 3091,
3090, 3032, 3031, 2973, 2972, 2916, 2915, 2857, 2856, 2798, 2797, 2707, 2650, 2649, 2591, 2590,
2532, 2531, 2464, 2463, 2407, 2406, 2348, 2347, 2289, 2220, 2219, 2161, 2160, 2102, 2101, 2043,
2042, 1984, 1983, 1925, 1924, 1865, 1782, 1781, 1671, 1670, 1564, 1563, 1089, 1088, 1032, 1031,
973, 972, 916, 915, 859, 858, 803, 802, 483, 482, 164, 163, 54, 53, 52, 51, 49, 48, 47, 46, 45,
44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21,
20, 19, 18, 17, 16, 15, 14, 13, 12, 9, 8, 7, 6, 5, 4, 3, 2, 11, 1, 10, 0]
{'pid': 0, 'name': 'kernel'}
{'pid': 1, 'name': 'init'}
{...}
psutil.boot_time() = 1570456872.0
2019-10-07 07:01:12
psutil.users() = [suser(name='user', terminal='pts/0', host='198.51.100.1',
started=1570552539.0)]
```

RELATED DOCUMENTATION

Overview of Python Modules on Junos Devices | 280

Requirements for Executing Python Automation Scripts on Junos Devices | 276

How to Use the Requests Library for Python on Devices Running Junos OS

IN THIS SECTION

The Requests library for Python is available on certain devices running Junos OS that support the Python extensions package. You can use the `requests` module in Python scripts to send HTTP/1.1 requests. On devices running Junos OS with Enhanced Automation, you can also use the `requests` module in Python interactive mode. The Requests library provides additional methods for supporting initial deployments as well as for performing routine monitoring and configuration changes on devices running Junos OS. For information about the `requests` module and its functions, see the Requests documentation at http://docs.python-requests.org/.

Issuing Requests

You can use the `requests` module in onbox Python scripts to send HTTP/1.1 requests. To make a request, import the module in your script, and call the function corresponding to the desired request. The module supports HTTP `GET` and `POST` requests as well as `HEAD`, `DELETE`, and `PUT` requests. The request returns a *Response* object containing the server's response. By default, requests are made using the default routing instance.

The Requests library can be used to execute RPCs on devices running Junos OS that support the REST API service. The target device must be configured with the appropriate statements at the `[edit system services rest]` hierarchy level to enable Junos OS commands over HTTP or HTTPS using REST.

For example, the following op script performs a GET request that executes the `get-software-information` RPC on a remote device running Junos OS that has the REST API service over HTTP configured on the default port (3000). The script prints the response status code, and if the status code indicates success, it prints the response content.

```
from junos import Junos_Context
import jcs
import requests

user = Junos_Context['user-context']['user']
password = jcs.get_secret('Enter user password: ')

r = requests.get('http://198.51.100.1:3000/rpc/get-software-information', auth=(user, password))
print (r.status_code)
if (r.status_code == requests.codes.ok):
    print (r.text)
```

```
user@host> op show-version.py
Enter user password:
200
<software-information>
<host-name>router1</host-name>
<product-model>mx240</product-model>
<product-name>mx240</product-name>
<junos-version>18.3R1.8</junos-version>
<package-information>
<name>os-kernel</name>
...
```

To retrieve just the headers, you can send a simple `HEAD` request.

```
r = requests.head('http://198.51.100.1:3000/rpc/get-software-information', auth=(user, password))
print (r.headers)
print (r.headers['content-type'])
```

```
user@host> op request-headers.py
Enter user password:
{'Date': 'Tue, 02 Apr 2019 18:30:58 GMT', 'Connection': 'close', 'Content-Type': 'application/
xml; charset=utf-8', 'Server': 'lighttpd/1.4.48'}
application/xml; charset=utf-8
```

If a GET request requires additional parameters, you can either include the `params` argument and supply a dictionary or a list of tuples or bytes to send in the query string, or you can pass in key/value pairs as part of the URL. Similarly, you can supply custom headers by including the `headers` argument and a dictionary of HTTP headers.

The following request executes the `get-interface-information` RPC with the terse option for the given interface and returns the response in text format:

```
headers={'content-type': 'application/xml', 'Accept': 'text/plain'}
params={'interface-name':'ge-2/0/1', 'terse':''}

r = requests.get('http://198.51.100.1:3000/rpc/get-interface-information', auth=(user,
password), headers=headers, params=params)
```

The following example supplies the arguments as key/value pairs in the URL:

```
headers={'content-type': 'application/xml', 'Accept': 'text/plain'}
rpc = 'get-interface-information?interface-name=ge-2/0/1&terse='

r = requests.get('http://198.51.100.1:3000/rpc/' + rpc, auth=(user, password), headers=headers)
```

To execute multiple RPCs in the same request, initiate an HTTP POST request, and set the `data` parameter to reference the RPCs to execute. See sections "Executing Operational RPCs" on page 297 and "Managing the Configuration" on page 300 for examples that execute multiple RPCs.

Executing Operational RPCs

You can use the `requests` module to execute RPCs from the Junos XML API on a remote device running Junos OS that has the REST API service enabled.

The following op script uses the `requests` module to execute the RPC equivalent of the `show interfaces ge-2/0/1 terse` operational mode command on the target device:

```
from junos import Junos_Context
import jcs
import requests

user = Junos_Context['user-context']['user']
password = jcs.get_secret('Enter user password: ')

r = requests.get('http://198.51.100.1:3000/rpc/get-interface-information', auth=(user,
```

```
password), params={'interface-name':'ge-2/0/1','terse':''})
print(r.text)
```

The following op script sends a POST request that executes multiple RPCs on the target device. The `data` parameter references the RPCs to execute, which are defined in a multiline string for readability.

```
from junos import Junos_Context
import jcs
import requests

user = Junos_Context['user-context']['user']
password = jcs.get_secret('Enter user password: ')

headers={'content-type': 'application/xml', 'Accept': 'text/plain'}

payload="""
<get-software-information/>
<get-interface-information>
  <interface-name>ge-2/0/1</interface-name>
</get-interface-information>"""

r = requests.post('http://198.51.100.1/rpc/', auth=(user, password), headers=headers,
data=payload)
if (r.status_code == requests.codes.ok):
    print (r.text)
```

You can also create a generic op script for which the user supplies the necessary variables and the script constructs and executes the request. Consider the following op script configuration, which configures the `host`, `rpc`, and `rpc_args` command line arguments for the **requests-rpc.py** op script:

```
[edit system scripts]
op {
    file requests-rpc.py {
        arguments {
            host {
                description "host:port to which to connect";
            }
            rpc {
                description "base RPC to execute on target device";
            }
            rpc_args {
```

```
                description "dictionary of RPC arguments to use";
            }
        }
    }
}
language python;
```

The following sample op script connects to a remote device running Junos OS, which has been configured with the appropriate statements at the `[edit system services rest]` hierarchy level to enable Junos OS commands over HTTP using REST. The script prompts for the connection password and connects to the host and port provided through the `host` argument. The script then uses the `requests` module to send a GET request executing the RPC that was provided through the command-line arguments.

NOTE: Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes command-line arguments to a Python op script, it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names. In earlier releases, the devices prefixes a single hyphen (-) to all argument names.

```
# Junos OS Release 21.1 and earlier
from junos import Junos_Context
from ast import literal_eval
import jcs
import argparse
import requests

## Argument list as configured in [edit system scripts op]
arguments = {  'host': 'host:port to which to connect',
               'rpc' : 'base RPC to execute on target device',
               'rpc_args' : 'dictionary of RPC arguments to use'
}

## Retrieve script arguments (Junos OS Release 21.1 and earlier)
parser = argparse.ArgumentParser(description='This is a demo script.')
for key in arguments:
    if key == 'rpc_args':
        parser.add_argument(('-' + key), help=arguments[key])
    else:
        parser.add_argument(('-' + key), required=True, help=arguments[key])
args = parser.parse_args()
```

```
## Convert rpc_args to a dictionary
if args.rpc_args is not None:
    args.rpc_args = literal_eval(args.rpc_args)

## Retrieve username and prompt for password for connecting to target device
user = Junos_Context['user-context']['user']
password = jcs.get_secret('Enter user password: ')

## Execute RPC
if args.rpc_args is None:
    r = requests.get('http://' + args.host + '/rpc/' + args.rpc, auth=(user, password))
else:
    r = requests.get('http://' + args.host + '/rpc/' + args.rpc, auth=(user, password),
params=args.rpc_args)

## Print RPC contents if HTTP status code indicates success
if (r.status_code == requests.codes.ok):
    print (r.text)
else:
    print (r.status_code)
```

When you execute the script, it executes the RPC with the specified options on the remote device and prints the response to standard output.

```
user@host> op requests-rpc.py host 198.51.100.1:3000 rpc get-interface-information rpc_args
{'interface-name':'ge-2/0/1','terse':''}
Enter user password:

<interface-information xmlns="http://xml.juniper.net/junos/18.3R1/junos-interface"
xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:style="terse">
<physical-interface>
<name>ge-2/0/1</name>
<admin-status>up</admin-status>
<oper-status>up</oper-status>
</physical-interface>
</interface-information>
```

Managing the Configuration

You can use the `requests` module to retrieve or change the configuration on a device running Junos OS that has the REST API service enabled.

The following op script retrieves the `[edit system]` hierarchy from the candidate configuration using a POST request:

```
from junos import Junos_Context
import jcs
import requests

user = Junos_Context['user-context']['user']
password = jcs.get_secret('Enter user password: ')

headers = { 'content-type' : 'application/xml' }
payload = '<get-configuration><configuration><system/></configuration></get-configuration>'

r = requests.post('http://198.51.100.1:3000/rpc/', auth=(user, password), data=payload,
headers=headers)
print (r.content)
```

HTTP POST requests also enable you to execute multiple RPCs in a single request, for example, to lock, load, commit, and unlock a configuration.

The following sample op script connects to the remote device and configures an address on the given interface. The lock, load, commit, and unlock operations are defined separately for readability, but the RPCs are concatenated in the request.

```
from junos import Junos_Context
import jcs
import requests

user = Junos_Context['user-context']['user']
password = jcs.get_secret('Enter user password: ')

#### lock, load, commit, unlock configuration
headers={'content-type': 'application/xml'}

lock = '<lock><target><candidate/></target></lock>'

load = """<edit-config>
  <target><candidate></candidate></target>
  <default-operation>merge</default-operation>
  <config>
    <configuration>
      <interfaces>
```

```
        <interface>
          <name>ge-2/0/1</name>
          <unit>
            <name>0</name>
            <family>
              <inet>
               <address>
                  <name>192.0.2.1/24</name>
                </address>
              </inet>
            </family>
          </unit>
        </interface>
      </interfaces>
    </configuration>
  </config>
  <error-option>stop-on-error</error-option>
</edit-config>"""

commit = '<commit/>'
unlock = '<unlock><target><candidate/></target></unlock>'

payload = lock + load + commit + unlock

r = requests.post('http://198.51.100.1:3000/rpc/', auth=(user, password), headers=headers,
data=payload)
print(r.content)
```

When you execute the op script, it returns the RPC results for the lock, load, commit, and unlock operations. On some devices, the response output separates the individual RPC replies with boundary lines that include -- followed by a boundary string and a `Content-Type` header. Other devices might include just the `Content-Type` header.

```
user@host> op requests-set-interface.py
Enter user password:
--harqgehabymwiax
Content-Type: application/xml; charset=utf-8

<ok/>
--harqgehabymwiax
Content-Type: application/xml; charset=utf-8
```

```
<load-success/>
--harqgehabymwiax
Content-Type: application/xml; charset=utf-8

<commit-results xmlns:junos="http://xml.juniper.net/junos/*/junos">
<routing-engine junos:style="normal">
<name>re0</name>
<commit-success/>
<commit-revision-information>
<new-db-revision>re0-1555351754-53</new-db-revision>
<old-db-revision>re0-1555033614-52</old-db-revision>
</commit-revision-information>
</routing-engine>
</commit-results>
--harqgehabymwiax
Content-Type: application/xml; charset=utf-8

<ok/>
--harqgehabymwiax--
```

Using Certificates in HTTPS Requests

The HTTP basic authentication mechanism sends user credentials as a Base64-encoded clear-text string. To protect the authentication credentials from eavesdropping, we recommend enabling the RESTful API service over HTTPS, which encrypts the communication using Transport Layer Security (TLS) or Secure Sockets Layer (SSL). For information about configuring this service, see the *Junos OS REST API Guide*.

By default, the Requests library verifies SSL certificates for HTTPS requests. You can include the `verify` and `cert` arguments in the request to control the SSL verification options. For detailed information about these options, see the Requests documentation.

NOTE: When you use Python 2.7 to execute a script that uses the `requests` module to execute HTTPS requests, the script generates `InsecurePlatformWarning` and `SubjectAltNameWarning` warnings.

The following op script sends a GET request over HTTPS, and sets the `verify` argument to the file path of a CA bundle or a directory containing trusted CA certificates. The specified CA certificates are used to verify the server's certificate.

```
from junos import Junos_Context
import jcs
```

```
import requests

user = Junos_Context['user-context']['user']
password = jcs.get_secret('Enter user password: ')

r = requests.get('https://198.51.100.1:3443/rpc/get-software-information', auth=(user,
password), verify='path-to-ca-bundle')
print (r.status_code)
if (r.status_code == requests.codes.ok):
    print (r.text)
```

To specify a local client-side certificate, set the `cert` argument equal to the path of a single file containing the client's private key and certificate or to a tuple containing the paths of the individual client certificate and private key files.

```
r = requests.get('https://198.51.100.1:3443/rpc/get-software-information', auth=(user,
password), verify='path-to-ca-bundle', cert=('path-to-client-cert','path-to-client-key')
```

Specifying the Routing Instance

By default, requests are executed using the default routing instance. You can also execute requests using the `mgmt_junos` management instance or another non-default routing instance. When you execute scripts through the Junos OS infrastructure, you can specify the routing instance by calling the `set_routing_instance()` function in the script. Certain devices also support specifying the routing instance and executing a script in the Unix-level shell.

NOTE: On devices running Junos OS Evolved, the `set_routing_instance()` function only supports using the management routing instance.

In a Python script, to execute a request using a non-default routing instance, including the `mgmt_junos` instance:

1. Import the `jcs` module.

```
import jcs
```

2. Call the `set_routing_instance()` function, and specify the instance to use for the connection.

```
jcs.set_routing_instance('mgmt_junos')
```

3. Establish the connection with the target device.

The following op script uses the `mgmt_junos` management instance to connect to the target device and execute requests.

```
from junos import Junos_Context
import jcs
import requests

user = Junos_Context['user-context']['user']
password = jcs.get_secret('Enter user password: ')

jcs.set_routing_instance('mgmt_junos')
r = requests.get('http://198.51.100.1:3000/rpc/get-software-information', auth=(user, password))
print (r.text)
```

For information about using the `set_routing_instance()` function in Python scripts, see "set_routing_instance()" on page 412.

In addition to specifying the routing instance in the script, certain devices support specifying the routing instance and executing a script from the Unix-level shell. On devices running Junos OS with Enhanced Automation (FreeBSD Release 7.1 or later), you can use the `setfib` command to execute requests with the given routing instance, including the management instance and other non-default routing instances.

The following Python script simply executes the `get-software-information` RPC on a remote device and prints the response:

```
#!/usr/bin/env python
import requests
from getpass import getpass

user =  raw_input ('Enter username: ')
password = getpass('Enter password: ')

r = requests.get('http://198.51.100.1:3000/rpc/get-software-information', auth=(user, password))
print (r.text)
```

To use `setfib` to execute the script using a non-default routing instance on a device running Junos OS with Enhanced Automation:

1. Find the software index associated with the routing table for that instance.

In the following example, the device is configured to use the non-default dedicated management instance `mgmt_junos`. The routing table index is referenced in the command output.

```
user@host> show route forwarding-table extensive table mgmt_junos
Routing table: mgmt_junos.inet [Index 36738]
Internet:
Enabled protocols: Bridging,

Destination:  default
  Route type: permanent
  Route reference: 0                   Route interface-index: 0
  Multicast RPF nh index: 0
  P2mpidx: 0
  Flags: none
  Next-hop type: reject                Index: 340      Reference: 1
```

To execute the op script with the given routing instance, use the `setfib` command to execute the script and reference the index. For example:

```
user@host> start shell
% setfib -F36738 python /var/db/scripts/op/request-software-info.py
```

In the following example, the device is configured with a non-default routing instance, vr1, and the vr1.inet routing table index is 8:

```
user@host> show route forwarding-table extensive table vr1
Routing table: vr1.inet [Index 8]
Internet:
Enabled protocols: Bridging, All VLANs,

Destination:  default
  Route type: permanent
  Route reference: 0                   Route interface-index: 0
  Multicast RPF nh index: 0
  P2mpidx: 0
  Flags: sent to PFE
  Next-hop type: reject                Index: 592      Reference: 1
```

The following command executes the op script using the vr1 routing instance:

```
% setfib -F8 python /var/db/scripts/op/request-software-info.py
```

Performing ZTP Operations

Zero touch provisioning (ZTP) enables you to provision new Juniper Networks devices in your network automatically, with minimal manual intervention. To use ZTP, you configure a server to provide the required information, which can include a Junos OS image and a configuration file to load or a script to execute. When you physically connect a device to the network and boot it with a factory-default configuration, the device retrieves the information from the designated server, upgrades the Junos OS image as appropriate, and executes the script or loads the configuration file.

When you connect and boot a new networking device, if Junos OS detects a file on the server, the first line of the file is examined. If Junos OS finds the characters #! followed by an interpreter path, it treats the file as a script and executes it with the specified interpreter. You can use the Requests library in executed scripts to streamline the ZTP process.

For example, consider the following sample Python script, which the new device downloads and executes during the ZTP process. When the script executes, it first downloads the CA certificate from the `ca_cert_remote` location on the specified server and stores it locally in the `ca_cert_local` location. The script then connects to the configuration server on port 8000 and issues a GET request to retrieve the new device configuration. The request includes the path to the CA certificate, which is used to verify the server's certificate during the exchange. The script then uses the Junos PyEZ library to load the configuration on the device and commit it.

```
#!/usr/bin/python
import os
import paramiko
import requests
from jnpr.junos import Device
from jnpr.junos.utils.config import Config

# Define the servers storing the certificate and configuration
host_cert = '198.51.100.1'
host_config = '192.0.2.1'
username = 'admin'
password = 'secret123'

# Define CA certificate file locations
ca_cert_remote = '/u01/app/myCA/certs/rootCA.crt'
ca_cert_local = '/var/tmp/rootCA.crt'
```

```
# Retrieve the CA certificate from the server
ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh.connect(hostname=host_cert, username=username, password=password)
sftp = ssh.open_sftp()
sftp.get(ca_cert_remote, ca_cert_local)
sftp.close()
ssh.close()

# Retrieve the configuration from the server
uri = 'https://' + host_config + ':8000/'
config = requests.get(uri, auth=(username, password), verify=ca_cert_local)

# Load and commit the configuration on the device
with Device() as dev:
    cu = Config(dev)
    cu.load(config.text, format='text', overwrite=True)
    cu.commit()
```

RELATED DOCUMENTATION

Overview of Python Modules on Junos Devices | 280

IPv6 Support in Python Automation Scripts

Starting in Junos OS Release 19.3R1, devices running Junos OS with upgraded FreeBSD support using IPv6 in Python automation scripts, including commit, event, op, and SNMP scripts, Juniper Extension Toolkit (JET) scripts, and YANG action scripts. IPv6 support enables Python automation scripts to establish connections and perform operations using IPv6 addresses.

For example, the following op script uses Junos PyEZ to connect to the host at the specified IPv6 address and print the device hostname and information about the ge-0/0/0 interface. The script retrieves the username from the script inputs and prompts for the user's password.

```
from jnpr.junos import Device
from lxml import etree
from junos import Junos_Context
import jcs
```

```
user = Junos_Context['user-context']['user']
password = jcs.get_secret('Enter user password: ')

try:
    with Device(host='2001:db8::2', user=user, password=password) as dev:
        print (dev.facts['hostname'])
        int_info = dev.rpc.get_interface_information({'format':'text'},
                   interface_name='ge-0/0/0', terse=True)
        print (etree.tostring(int_info, encoding='unicode'))
except Exception as err:
    print (err)
```

To execute the script, configure the `language python` or `language python3` statement and the script filename.

```
[edit]
user@host# set system scripts language python3
user@host# set system scripts op file connect-ipv6.py
user@host# commit and-quit
```

When you execute the script, it prints the hostname of the remote device and information about the ge-0/0/0 interface.

```
user@host> op connect-ipv6.py
Enter user password:
r2
<output>
Interface               Admin Link Proto    Local                 Remote
ge-0/0/0                up    up
ge-0/0/0.0              up    up   inet6    2001:db8::2/32
                                            fe80::5668:a3ff:fe16:79ef/64
                                   multiservice
</output>
```

How to Specify the Routing Instance in Python 3 Applications on Devices Running Junos OS Evolved

IN THIS SECTION

Understanding the libpyvrf Module

A routing instance is a collection of routing tables, interfaces, and routing protocol parameters. Each routing instance has a unique name and a corresponding IP unicast table, which can be used to separate traffic for that instance from other traffic. For example, on devices running Junos OS Evolved, you can enable a dedicated management virtual routing and forwarding (VRF) instance, `mgmt_junos`, which uses a separate administrative routing table dedicated to management tasks.

Routing instances enable you to isolate traffic traversing the network without using multiple devices to segment your network. When you use routing instances to isolate traffic, network utilities like `ping` and `ssh` must reference the correct routing instance in order to send traffic to a destination that is reachable through that instance.

The Junos OS Evolved image runs natively on Linux, providing direct access to all the Linux utilities and operations, including the Python libraries that are part of the base image. The Python 3 library on devices running Junos OS Evolved also includes the Juniper Networks `libpyvrf` module, which enables Python 3 applications that are executed in the shell to specify the routing instance to use for specific processes and network utilities.

Table 33 on page 311 outlines the `libpyvrf` functions that you can use in Python 3 applications that are executed in the Linux shell. You can use the functions to instruct a process to use a specific routing instance. If the same process requires multiple routing instances, you can also set the routing instance context for a specific socket, which does not affect the context for the process itself. Packets through the socket then use the routing table associated with that instance. When you set the routing instance for a process or socket, the `libpyvrf` module sets the context to the Linux VRF that corresponds to the Junos OS routing instance.

Table 33: libpyvrf Functions

`libpyvrf` Function	Description
`get_host_vrf_name()`	Return the Linux VRF corresponding to a Junos OS routing instance.
`get_table_id()`	Return the index of the routing table associated with the specified routing instance.
`get_task_vrf()`	Return the routing instance associated with the specified task. **NOTE**: This function is deprecated starting in Junos OS Evolved Release 22.1R1.
`get_vrf()`	Return the routing instance associated with the current process.
`set_socket_vrf()`	Set the routing instance for the specified socket.
`set_task_vrf()`	Set the routing instance for the specified process. **NOTE**: This function is deprecated starting in Junos OS Evolved Release 22.1R1.
`set_vrf()`	Set the routing instance for the current process.

Functions in the `libpyvrf` module can raise the following exceptions depending on the function and error:

- `libpyvrf.error`—Generated when `libvrf` returns an error.
- `libpyvrf.evo_not_ready`—Generated when the Junos OS Evolved network stack is not ready, for example when the device is booting.
- `libpyvrf.invalid_table`—Generated when the specified routing instance or table ID is invalid.

The following sample Python script attempts to ping a host that is only reachable through the `mgmt_junos` routing instance. The script initially pings the host before calling the `set_vrf()` function. The script then calls the `set_vrf()` function to associate the `mgmt_junos` routing instance with the current process and pings the host again.

```
[vrf:none] user@host:~# cat libpyvrf-ping.py
import libpyvrf as vrf
```

```
import subprocess

command = [ 'ping', '-c', '3', 'host1.example.com' ]

try:
    # Ping the host before setting the routing instance
    subprocess.call(command) == 0

    # Set the routing instance
    vrf.set_vrf("mgmt_junos")
    print ("\nUsing routing instance:", vrf.get_vrf())

    # Ping the host after setting the routing instance
    subprocess.call(command) == 0

except vrf.invalid_table as e:
    print ("Invalid Table")
except vrf.evo_not_ready as e :
    print ("Junos OS Evolved network stack is not ready")
except vrf.error as e :
    print ("Generic libvrf error")
```

When you execute the script, the first ping command fails, because the process uses the default routing instance in this case, and the host is only reachable through the `mgmt_junos` routing instance. The second ping command, which uses the `mgmt_junos` routing instance, succeeds.

```
[vrf:none] user@host:~# python3 libpyvrf-ping.py
ping: unknown host host1.example.net

Using routing instance: mgmt_junos
PING host1.example.com (198.51.100.10) 56(84) bytes of data.
64 bytes from host1.example.com (198.51.100.10): icmp_seq=1 ttl=60 time=1.02 ms
64 bytes from host1.example.com (198.51.100.10): icmp_seq=2 ttl=60 time=0.672 ms
64 bytes from host1.example.com (198.51.100.10): icmp_seq=3 ttl=60 time=0.741 ms

--- host1.example.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.876/0.946/1.001/0.063 ms
```

You can reset the routing instance back to the default routing instance association in a Python application by specifying an empty string for the routing instance name. For example:

```
vrf.set_vrf("")
```

If an application does not specify a routing instance, the default routing instance association depends on the environment of the process. If the environment preloads the **libsi.so** library, then by default, the application uses the default routing instance and table, `default.inet`. Otherwise, there is no default routing instance associated with the process or sockets.

To determine if the environment preloads the **libsi.so** library, verify that value of the `LD_PRELOAD` environment variable includes the path to the library.

```
[vrf:none] user@host:~# env | grep LD_PRELOAD
LD_PRELOAD=libsi.so
```

If the `LD_PRELOAD` variable does not include the **libsi.so** library path, you can use the commands appropriate for your shell to add it, for example:

```
[vrf:none] user@host:~# export LD_PRELOAD="/path/to/libsi.so"
[vrf:none] user@host:~# env | grep LD_PRELOAD
LD_PRELOAD=libsi.so
```

get_host_vrf_name() Function

IN THIS SECTION

Syntax

```
host_vrf_name get_host_vrf_name(vrf_name)
```

Description

Return the Linux VRF corresponding to a Junos OS routing instance.

Parameters

vrf_name	Name of a Junos OS routing instance.

Return Value

host_vrf_name	Name of the Linux VRF corresponding to the Junos OS routing instance.

Usage Examples

The following example prints the Linux VRF corresponding to several Junos OS routing instances:

```
[vrf:none] user@host:~# cat libpyvrf-get-host-vrf-name.py
import libpyvrf as vrf

try:
    print ("mgmt_junos VRF:", vrf.get_host_vrf_name("mgmt_junos"))
    print ("default VRF:", vrf.get_host_vrf_name("default"))
    print ("L3VPN-1 VRF:", vrf.get_host_vrf_name("L3VPN-1"))
except vrf.invalid_table as e:
    print ("Invalid Table")
```

```
[vrf:none] user@host:~# python3 libpyvrf-get-host-vrf-name.py
mgmt_junos VRF: mgmt_junos
default VRF: vrf0
L3VPN-1 VRF: vrf52
```

Release Information

Function introduced in Junos OS Evolved Release 20.3R1.

get_table_id() Function

IN THIS SECTION

Syntax

```
table_id get_table_id(vrf_name)
```

Description

Return the index of the routing table associated with the specified routing instance. If the routing instance is not defined, the function returns -1.

Parameters

vrf_name	Name of a Junos OS routing instance.

Return Value

table_id	Index of the routing table for the given routing instance.

Usage Examples

The following example retrieves and prints the table index for the `mgmt_junos` routing instance and an undefined routing instance `foo`.

```
user@host> show route forwarding-table extensive table mgmt_junos

Routing table: mgmt_junos.inet [Index 36738]
Internet:
...
Routing table: mgmt_junos.inet6 [Index 36738]
Internet6:
...
```

```
[vrf:none] user@host:~# cat libpyvrf-get-table-id.py
import libpyvrf as vrf

try:
    print (vrf.get_table_id("mgmt_junos"))
    print (vrf.get_table_id("foo"))
except vrf.evo_not_ready as e:
    print ("Junos OS Evolved network stack is not ready")
except vrf.error as e:
    print ("Generic libvrf error")
```

The script returns the table index for the `mgmt_junos` instance and returns -1 for the undefined instance.

```
[vrf:none] user@host:~# python3 libpyvrf-get-table-id.py
36738
-1
```

Release Information

Function introduced in Junos OS Evolved Release 20.3R1.

get_task_vrf() Function

IN THIS SECTION

Syntax

```
vrf_name get_task_vrf(tid)
```

Description

Return the Junos OS routing instance associated with the specified process ID.

Parameters

tid Process ID for which to retrieve the associated routing instance.

Return Value

vrf_name Name of the routing instance associated with the process ID.

Usage Examples

The following example retrieves the process ID for the current process and associates the `mgmt_junos` routing instance with that process. When the script calls the `get_task_vrf()` function to request the routing instance for that process ID, it returns the `mgmt_junos` routing instance.

```
[vrf:none] user@host:~# cat libpyvrf-set-task-vrf.py
import libpyvrf as vrf
import os, subprocess

command = [ 'ping', '-c', '3', 'host1.example.com' ]

try:
    pid = os.getpid()
    vrf.set_task_vrf(pid, "mgmt_junos")
    print ("Using routing instance:", vrf.get_task_vrf(pid))
    subprocess.call(command) == 0
except vrf.invalid_table as e:
    print ("Invalid Table")
except vrf.evo_not_ready as e:
    print ("Junos OS Evolved network stack is not ready")
except vrf.error as e:
    print ("Generic libvrf error")
```

```
[vrf:none] user@host:~# python3 libpyvrf-set-task-vrf.py
Using routing instance: mgmt_junos
PING host1.example.com (198.51.100.10) 56(84) bytes of data.
64 bytes from host1.example.com (198.51.100.10): icmp_seq=1 ttl=60 time=1.02 ms
64 bytes from host1.example.com (198.51.100.10): icmp_seq=2 ttl=60 time=0.672 ms
64 bytes from host1.example.com (198.51.100.10): icmp_seq=3 ttl=60 time=0.741 ms

--- host1.example.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.876/0.946/1.001/0.063 ms
```

Release Information

Function introduced in Junos OS Evolved Release 20.3R1.

Function deprecated in Junos OS Evolved Release 22.1R1.

get_vrf() Function

IN THIS SECTION

Syntax

```
vrf_name get_vrf()
```

Description

Return the Junos OS routing instance associated with the current process.

Return Value

vrf_name	Name of the routing instance associated with the current process.

Usage Examples

The following example associates the `mgmt_junos` routing instance with the current process. When the script calls the `get_vrf()` function to request the routing instance for the current process, it returns the `mgmt_junos` routing instance.

```
[vrf:none] user@host:~# cat libpyvrf-set-vrf.py
import libpyvrf as vrf
import subprocess

command = [ 'ping', '-c', '3', 'host1.example.com' ]

try:
```

```
    vrf.set_vrf("mgmt_junos")
    print ("Using routing instance:", vrf.get_vrf())
    subprocess.call(command) == 0
except vrf.invalid_table as e:
    print ("Invalid Table")
except vrf.evo_not_ready as e:
    print ("Junos OS Evolved network stack is not ready")
except vrf.error as e:
    print ("Generic libvrf error")
```

```
[vrf:none] user@host:~# python3 libpyvrf-set-vrf.py
Using routing instance: mgmt_junos
PING host1.example.com (198.51.100.10) 56(84) bytes of data.
64 bytes from host1.example.com (198.51.100.10): icmp_seq=1 ttl=60 time=1.02 ms
64 bytes from host1.example.com (198.51.100.10): icmp_seq=2 ttl=60 time=0.672 ms
64 bytes from host1.example.com (198.51.100.10): icmp_seq=3 ttl=60 time=0.741 ms

--- host1.example.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.876/0.946/1.001/0.063 ms
```

Release Information

Function introduced in Junos OS Evolved Release 20.3R1.

set_socket_vrf() Function

IN THIS SECTION

Syntax

```
set_socket_vrf(socket_fd, vrf_name)
```

Description

Set the routing instance used by the specified socket. Setting the routing instance for a socket is useful when multiple sockets within the same application need to use different routing instances. You can set the routing instance context for each socket individually without affecting the routing instance context for the process or application.

Parameters

socket_fd Socket's file descriptor.

vrf_name Name of a Junos OS routing instance.

- **Values:** Acceptable values include `"default"`, `"iri"`, `"mgmt_junos"`, or the name defined for any user-configured routing instance in the Junos OS configuration.

Release Information

Function introduced in Junos OS Evolved Release 20.3R1.

set_task_vrf() Function

IN THIS SECTION

Syntax

```
set_task_vrf(tid, vrf_name)
```

Description

Set the routing instance that the process with the specified process ID will use to perform operations.

Whereas `set_vrf()` sets the routing instance for the current process, `set_task_vrf()` sets the routing instance for the process with the specified process ID.

Parameters

tid Process ID for the process that will use the specified routing instance.

vrf_name Name of a Junos OS routing instance.

- **Values:** Acceptable values include `"default"`, `"iri"`, `"mgmt_junos"`, or the name defined for any user-configured routing instance in the Junos OS configuration.

Usage Examples

The following sample Python script retrieves the process ID for the current process and associates the `mgmt_junos` routing instance with that process. The script then pings a host that is only reachable through that routing instance.

```
[vrf:none] user@host:~# cat libpyvrf-set-task-vrf.py
import libpyvrf as vrf
import os, subprocess

command = [ 'ping', '-c', '3', 'host1.example.com' ]

try:
    pid = os.getpid()
    vrf.set_task_vrf(pid, "mgmt_junos")
    print ("Using routing instance:", vrf.get_task_vrf(pid))
    subprocess.call(command) == 0
except vrf.invalid_table as e:
    print ("Invalid Table")
except vrf.evo_not_ready as e:
    print ("Junos OS Evolved network stack is not ready")
```

```
except vrf.error as e:
    print ("Generic libvrf error")
```

```
[vrf:none] user@host:~# python3 libpyvrf-set-task-vrf.py
Using routing instance: mgmt_junos
PING host1.example.com (198.51.100.10) 56(84) bytes of data.
64 bytes from host1.example.com (198.51.100.10): icmp_seq=1 ttl=60 time=1.02 ms
64 bytes from host1.example.com (198.51.100.10): icmp_seq=2 ttl=60 time=0.672 ms
64 bytes from host1.example.com (198.51.100.10): icmp_seq=3 ttl=60 time=0.741 ms

--- host1.example.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.876/0.946/1.001/0.063 ms
```

Release Information

Function introduced in Junos OS Evolved Release 20.3R1.

Function deprecated in Junos OS Evolved Release 22.1R1.

set_vrf() Function

IN THIS SECTION

Syntax

```
set_vrf(vrf_name)
```

Description

Set the routing instance that the current process will use to perform operations. Future lookups will use this routing instance.

To associate a routing instance with a specific process instead of the current process, use the `set_task_vrf()` function.

Parameters

vrf_name Name of a Junos OS routing instance.

- **Values:** Acceptable values include `"default"`, `"iri"`, `"mgmt_junos"`, or the name defined for any user-configured routing instance in the Junos OS configuration.

Usage Examples

The following sample Python script associates the `mgmt_junos` routing instance with the current process. The script then pings a host that is only reachable through that routing instance.

```
import libpyvrf as vrf
import subprocess

command = [ 'ping', '-c', '3', 'host1.example.com' ]

try:
    vrf.set_vrf("mgmt_junos")
    print ("Using routing instance:", vrf.get_vrf())
    subprocess.call(command) == 0
except vrf.invalid_table as e:
    print ("Invalid Table")
except vrf.evo_not_ready as e:
    print ("Junos OS Evolved network stack is not ready")
except vrf.error as e:
    print ("Generic libvrf error")
```

```
[vrf:none] user@host:~# python3 libpyvrf-set-vrf.py
Using routing instance: mgmt_junos
PING host1.example.com (198.51.100.10) 56(84) bytes of data.
64 bytes from host1.example.com (198.51.100.10): icmp_seq=1 ttl=60 time=1.02 ms
64 bytes from host1.example.com (198.51.100.10): icmp_seq=2 ttl=60 time=0.672 ms
```

```
64 bytes from host1.example.com (198.51.100.10): icmp_seq=3 ttl=60 time=0.741 ms

--- host1.example.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.876/0.946/1.001/0.063 ms
```

Release Information

Function introduced in Junos OS Evolved Release 20.3R1.

5

PART

Automation Script Input

CHAPTER 10

Global Parameters in Automation Scripts

IN THIS CHAPTER

Global Parameters and Variables in Junos OS Automation Scripts

IN THIS SECTION

Junos OS automatically provides input to automation scripts when they are executed. The script can reference this input, which includes device-specific information about the script execution environment such as the device hostname, the script type, and the user executing the script. This information is useful for creating scripts that respond to a variety of complex scenarios.

SLAX and XSLT scripts that import the **junos.xsl** file can reference this information using the `$junos-context` global variable, which is a node-set. The **junos.xsl** import file also declares several predefined global parameters that enable the scripts to more easily reference a subset of this information. Python scripts can reference this information through the `junos.Junos_Context` dictionary, which must be imported into the script.

To use the pre-defined parameters or global variable in SLAX and XSLT scripts, you must import the **junos.xsl** file by including the `<xsl:import>` tag in the style sheet declaration of an *XSLT* script or by including the `import` statement in a SLAX script and specifying the **junos.xsl** file location as shown in the following sample code:

XSLT Syntax

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

SLAX Syntax

```
version 1.2;
import "../import/junos.xsl";
```

Python

To reference the information in Python scripts, import the `Junos_Context` dictionary.

```
from junos import Junos_Context
```

The script input is described in detail in the following sections:

Global Parameters Available in SLAX and XSLT Scripts

Several predefined global parameters are available for use in SLAX and XSLT automation scripts that import the **junos.xsl** file. The parameters provide information about the Junos OS environment. Table 34 on page 328 describes the built-in arguments.

Table 34: Predefined Parameters Available in SLAX and XSLT Scripts

Name	Description	Example
`$hostname`	Hostname of the local device	Tokyo
`$localtime`	Local time when the script is executed	Fri Dec 10 11:42:21 2010
`$localtime-iso`	Local time, in ISO format, when the script is executed	2010-12-10 11:42:21 PST
`$product`	Model of the local device	m10i

Table 34: Predefined Parameters Available in SLAX and XSLT Scripts *(Continued)*

Name	Description	Example
`$script`	Filename of the executing script	**test.slax**
`$user`	Local name of the user executing the script	root

The predefined global parameters are declared in the **junos.xsl** file. You do not need to declare these parameters in a script in order to use them. Access the value of the global parameters in a script by prefixing the parameter name with the dollar sign ($), as shown in the following example:

SLAX syntax:

```
if ($user != "root") {
    var $script-message = $user _ " does not have permission to execute " _ $script;
    expr jcs:output($script-message);
}
```

XSLT syntax:

```
<xsl:if test="$user != 'root'">
    <xsl:variable name="script-message"
     select="concat($user, ' does not have permission to execute ', $script)"/>
    <xsl:value-of select="jcs:output($script-message)"/>
</xsl:if>
```

Global Variable Available in Automation Scripts

Commit, event, and op scripts can access specific environment information that is provided to the script upon execution. To access this information, Python scripts must import and reference the `junos.Junos_Context` dictionary, and SLAX and XSLT scripts that import the **junos.xsl** file can reference the `$junos-context` global variable. `$junos-context` and `Junos_Context` contain identical information but in a format that is suitable for the respective script language.

The `$junos-context` variable is a node-set that contains the `<junos-context>` node and the following hierarchy, which is common to and embedded in the source tree of all scripts:

```
<junos-context>
    <chassis></chassis>
    <hostname></hostname>
    <localtime></localtime>
    <localtime-iso></localtime-iso>
    <pid></pid>
    <product></product>
    <re-master/>
    <routing-engine-name></routing-engine-name>
    <sw-upgrade-in-progress></sw-upgrade-in-progress>
    <script-type></script-type>
    <tty></tty>
    <user-context>
        <class-name></class-name>
        <login-name></login-name>
        <uid></uid>
        <user></user>
    </user-context>
</junos-context>
```

Additionally, script-specific information is available depending on the type of script executed. For op scripts, the `<op-context>` element is also included in the source tree provided to an *op script*.

```
<junos-context>
    <op-context>
        <via-url/>
    </op-context>
</junos-context>
```

For commit scripts, the `<commit-context>` element is also included in the source tree provided to a commit script.

```
<junos-context>
    <commit-context>
        <commit-comment>"This is a test commit"</commit-comment>
        <commit-boot/>
        <commit-check/>
        <commit-sync/>
```

```
        <commit-confirm/>
        <database-path/>
    </commit-context>
</junos-context>
```

Table 35 on page 331 identifies each node of the `$junos-context` variable node-set, provides a brief description of the node, and gives examples of values for any elements that are not input to a script as an empty tag.

Table 35: Global Variable $junos-context Available to SLAX and XSLT Scripts

Parent Node	Node	Description	Example Content
`<junos-context>`	`<chassis>`	Specifies whether the script is executed on a component of a *routing matrix*, the *Root System Domain* (RSD), or a *Protected System Domain* (PSD)	scc, lcc (TX Matrix) psd, rsd (JCS) others
	`<hostname>`	Hostname of the local device	Tokyo
	`<localtime>`	Local time when the script is executed	Fri Dec 10 11:42:21 2010
	`<localtime-iso>`	Local time, in ISO format, when the script is executed	2010-12-10 11:42:21 PST
	`<pid>`	cscript process ID	5257
	`<product>`	Model of the local device	m10i
	`<re-master/>`	Empty element included if the script is executed on the primary Routing Engine	–
	`<routing-engine-name>`	Routing Engine on which the script is executed	re0
	`<script-type>`	Type of script being executed	op

Table 35: Global Variable $junos-context Available to SLAX and XSLT Scripts *(Continued)*

Parent Node	Node	Description	Example Content
	`<sw-upgrade-in-progress>`	Element that enables a commit script to check if the commit occurs during the first reboot after a software install. The tag value is `yes` if the commit takes place during the first reboot after a software upgrade, software downgrade, or rollback. The tag value is `no` if the device is booting normally.	yes
	`<tty>`	TTY of the user's session	/dev/ttyp1
`<junos-context>` `<user-context>`	`<class-name>`	Login class of the user executing the script	superuser
	`<login-name>`	Login name of the user executing the script. For *AAA* access, this is the RADIUS/TACACS username.	jsmith
	`<uid>`	User ID of the user executing the script as defined in the device configuration	2999
	`<user>`	Local name of the user executing the script. Junos OS uses the local name for authentication. It might differ from the `login-name` used for AAA authentication.	root
`<junos-context>` `<op-context>` (op scripts only)	`<via-url>`	Empty element included if the remote op script is executed using the `op url` command	–
`<junos-context>` `<commit-context>` (commit scripts only)	`<commit-boot/>`	Empty element included when the commit occurs at boot time	–

Table 35: Global Variable $junos-context Available to SLAX and XSLT Scripts *(Continued)*

Parent Node	Node	Description	Example Content
	`<commit-check/>`	Empty element included when a `commit check` is performed	–
	`<commit-comment>`	User comment regarding the commit	Commit to fix forwarding issue
	`<commit-confirm/>`	Empty element included when a `commit confirmed` is performed	–
	`<commit-sync/>`	Empty element included when a `commit synchronize` is performed	–
	`<database-path/>`	Element specifying the location of the session's pre-inheritance candidate configuration. For normal configuration sessions, the value of the element is the location of the normal candidate database. For private configuration sessions, the value of the element is the location of the private candidate database. When the `<get-configuration>` `database-path` attribute is set to this value, the commit script retrieves the corresponding pre-inheritance candidate configuration.	–

The `$junos-context` variable is a node-set. Therefore, you can access the child elements throughout a script by including the proper *XPath* expression. The following SLAX commit script writes a message to the system log file if the commit is performed during initial boot-up. The message is given a *facility* value of `daemon` and a severity value of `info`. For more information, see "syslog()" on page 420.

```
version 1.2;
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 ($junos-context/commit-context/commit-boot) {
        expr jcs:syslog("daemon.info", "This is boot-time commit");
    }
    else {
         /* Do this ... */
    }
}
```

Python scripts must import the `Junos_Context` dictionary from the `junos` module to access the environment information provided to scripts. The names of the keys in the `Junos_Context` dictionary are identical to the names of the `$junos-context` nodes outlined in Table 35 on page 331. Nodes with child elements that are nested under the `junos-context` node such as `user-context`, `op-context`, and `commit-context` map to items in `Junos_Context`, where the key is the node name and the value is a dictionary of the node's child elements. For example:

```
'user-context': {'login-name': 'bsmith', 'user': 'bsmith', 'class-name': 'j-superuser', 'uid':
'9999'}
```

The following example output displays the `Junos_Context` dictionary for an op script that was executed locally. Note that the op script input contains the `op-context` key, which in this scenario is empty.

```
{'product': 'm7i', 'user-context': {'login-name': 'bsmith', 'user': 'bsmith', 'class-name': 'j-
superuser', 'uid': '9999'}, 'routing-engine-name': 're0', 'script-type': 'op', 're-master':
None, 'hostname': 'R1', 'pid': '7136', 'tty': '/dev/ttyp1', 'chassis': 'others', 'op-context':
' ', 'localtime': 'Thu Jan 22 11:45:47 2015', 'localtime-iso': '2015-01-22 11:45:47 PST'}
```

The following example output displays the `Junos_Context` dictionary for a commit script that was executed during a `commit check` operation. Note that the commit script input contains the `commit-context` key.

```
{'product': 'm7i', 'user-context': {'login-name': 'bsmith', 'user': 'bsmith', 'class-name': 'j-
superuser', 'uid': '9999'}, 'routing-engine-name': 're0', 'script-type': 'commit', 're-master':
None, 'hostname': 'R1', 'pid': '11201', 'tty': '/dev/ttyp1', 'commit-context': {'database-path':
'/var/run/db/juniper.db', 'commit-check': None}, 'chassis': 'others', 'localtime': 'Thu Jan 22
16:23:55 2015', 'localtime-iso': '2015-01-22 16:23:55 PST'}
```

To access individual values in the dictionary, specify the key name. For example:

```
Junos_Context['hostname']
Junos_Context['user-context']['class-name']
```

6

PART

Extension Functions and Named Templates for Automation Scripts

CHAPTER 11

Extension Functions for Automation Scripting

IN THIS CHAPTER

Understanding Extension Functions in Junos OS Automation Scripts

Junos OS provides automation scripting tools, including extension functions and named templates, that can be used in commit, op, event, and SNMP scripts to more easily accomplish scripting tasks on devices running Junos OS. The libraries provide logic, data manipulation, input and output, and utility functions and enable you to perform operations that are difficult or impossible to perform in *XPath*. Table 36 on page 338 summarizes the Juniper extension functions.

To use the extension functions, Python automation scripts must import the `jcs` module, and SLAX and XSLT scripts must declare the appropriate namespace URI in the style sheet declaration. A function in the `jcs` namespace is defined in the namespace URI http://xml.juniper.net/junos/commit-scripts/1.0, and a function in the `slax` namespace is defined in the namespace URI http://xml.libslax.org/slax .

Functions using the `slax` namespace are supported starting in Junos OS Release 12.2. Scripts using Junos OS-independent extension functions that existed in earlier releases in the `jcs` namespace can use either the `jcs` or the `slax` namespace starting in Junos OS Release 12.2. However, to use any of these functions in earlier Junos OS releases, scripts must use the `jcs` namespace URI.

The SLAX script `version` statement determines which functions can be used in that script. In order to use a function that was introduced in a specific SLAX version, the value of the script `version` statement must be equal to or greater than the version in which the function was introduced. For example, functions introduced in version 1.1 of the SLAX language cannot be used in a SLAX script that has a "version 1.0" statement.

NOTE: SLAX scripts can also use additional functions from the libslax default extension libraries. For more information, see "libslax Default Extension Libraries: bit, curl, db, os, and xutil" on page 224.

Python automation scripts only support a subset of the extension functions. For many of the unsupported functions, you can use standard Python libraries to achieve the same functionality.

Table 36: Extension Functions for Use in Automation Scripts

Function	Name-spaces	SLAX Version	Type	Description	Support in Python Scripts
`base64-decode()`	slax	1.1	Data manipulation	Decode BASE64 encoded data and return a string.	–
`base64-encode()`	slax	1.1	Data manipulation	Encode a string of data in the BASE64 encoding format.	–
`break-lines()`	jcs, slax	1.0	Data manipulation	Break a simple element into multiple elements, delimited by newlines.	–
`close()`	jcs	1.0	Utility	Close a previously opened connection handle.	–
`dampen()`	jcs, slax	1.0	Utility	Prevent the same operation from being repeatedly executed within a script.	Y
`document()`	slax	1.1	Input/output control	Read data from a file or URL and return a string.	–
`emit_change()`	–	–	Utility	Generate a persistent or transient change to the configuration in a commit script.	Y

Table 36: Extension Functions for Use in Automation Scripts *(Continued)*

Function	Name-spaces	SLAX Version	Type	Description	Support in Python Scripts
`emit_error()`	–	–	Input/output control	Generate an error message on the console.	Y
`emit_snmp_attributes()`	–	–	Input/output control	Return information for the requested MIB object from an SNMP script.	Y
`emit_warning()`	–	–	Input/output control	Generate a warning message on the console.	Y
`empty()`	jcs, slax	1.0	Logic	Evaluate a node set or string argument to determine if it is an empty value.	–
`evaluate()`	slax	1.1	Input/output control	Evaluate a SLAX expression and return the result.	–
`execute()`	jcs	1.0	Utility	Execute a *remote procedure call* (RPC) within the context of a specified connection handle.	–
`first-of()`	jcs, slax	1.0	Logic	Return the first nonempty (non-null) item in a list, If all objects in the list are empty, the default expression is returned.	–
`get-command()`	jcs, slax	1.1	Input/output control	Prompt the user for command input and return the input as a string.	–

Table 36: Extension Functions for Use in Automation Scripts *(Continued)*

Function	Name-spaces	SLAX Version	Type	Description	Support in Python Scripts
`get-hello()`	jcs	1.0	Utility	Return the session ID and the capabilities of the NETCONF server during a NETCONF session.	–
`get-input()` (XSLT/SLAX) `get_input()` (Python)	jcs, slax	1.0	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.	Y
`get-protocol()`	jcs	1.0	Utility	Return the session protocol associated with the connection handle.	–
`get-secret()` (XSLT/SLAX) `get_secret()` (Python)	jcs, slax	1.0	Input/output control	Invoke a CLI prompt and wait for user input. The input is not echoed back to the user.	Y
`get_snmp_action()`	–	–	Input/output control	Retrieve the action value passed to the SNMP script.	Y
`get_snmp_oid()`	–	–	Input/output control	Retrieve the OID value passed to the SNMP script.	Y
`hostname()`	jcs	1.0	Utility	Return the fully qualified domain name associated with a given IPv4 or IPv6 address, provided the *DNS* server is configured on the device.	Y

Table 36: Extension Functions for Use in Automation Scripts *(Continued)*

Function	Name-spaces	SLAX Version	Type	Description	Support in Python Scripts
`invoke()`	jcs	1.0	Utility	Invoke an RPC on a local device running Junos OS.	–
`open()`	jcs	1.0	Utility	Return a connection handle that can be used to execute RPCs.	–
`output()`	jcs, slax	1.0	Input/output control	Generate unformatted output text that is immediately sent to the CLI session.	Y
`parse-ip()`	jcs	1.0	Data manipulation	Parse an IPv4 or IPv6 address and return the host IP address, protocol family, prefix length, network address, and network mask.	Y
`printf()`	jcs, slax	1.0	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.	Y
`progress()`	jcs, slax	1.0	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.	Y

Table 36: Extension Functions for Use in Automation Scripts *(Continued)*

Function	Name-spaces	SLAX Version	Type	Description	Support in Python Scripts
`regex()`	jcs, slax	1.0	Data manipulation	Evaluate a regular expression against a given string argument and return any matches.	–
`set_routing_instance()`	-	-	Utility	Program the protocol software (TCP/UDP) to use nondefault routing instances.	Y
`sleep()`	jcs, slax	1.0	Utility	Cause the script to sleep for a specified time.	–
`split()`	jcs, slax	1.0	Data manipulation	Split a string into an array of substrings delimited by a regular expression pattern.	–
`sysctl()`	jcs, slax	1.0	Utility	Return the value of the given `sysctl` value as a string or an integer.	Y
`syslog()`	jcs, slax	1.0	Input/output control	Log messages with the specified priority to the system log file.	Y
`trace()`	jcs, slax	1.0	Input/output control	Issue a trace message, which is sent to the trace file.	Y

RELATED DOCUMENTATION

Use Extension Functions in Junos OS Automation Scripts | 343

Understanding Named Templates in Junos OS Automation Scripts | 427

Use Extension Functions in Junos OS Automation Scripts

IN THIS SECTION

Junos OS provides extension functions that can be used in commit, op, event, and SNMP scripts to more easily accomplish scripting tasks on devices running Junos OS. The following sections outline how to import and use the extension functions for different script languages:

Using Extension Functions in SLAX and XSLT Scripts

To use the extension functions in SLAX and XSLT scripts, the script must declare the appropriate namespace *Uniform Resource Identifier* (URI) in the style sheet declaration. Junos OS extension functions, which have functionality that is specific to devices running Junos OS, are defined in the namespace with the associated URI `http://xml.juniper.net/junos/commit-scripts/1.0`. SLAX extension functions are defined in the namespace with the associated URI `http://xml.libslax.org/slax`.

SLAX and XSLT scripts generally map the `jcs` or `slax` prefix to its respective URI to avoid name conflicts with standard *XSLT* functions and user-defined templates. The scripts then qualify the extension functions with the appropriate prefix, which is expanded into its associated URI reference during processing.

For example, the following SLAX namespace statement maps the `jcs` prefix to the namespace URI that defines the Junos OS extension functions used in automation scripts:

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

The following SLAX namespace statement maps the `slax` prefix to the namespace URI that defines SLAX extension functions:

```
ns slax = "http://xml.libslax.org/slax";
```

To call an extension function in a SLAX or XSLT script, include any required variable declarations, call the function using `jcs:`*`function-name`*`()` or `slax:`*`function-name`*`()` as appropriate, 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.

The following example maps the `jcs` prefix to the namespace identified by the URI `http://xml.juniper.net/junos/commit-scripts/1.0`. 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('get-software-information')"/>
    ...
</xsl: stylesheet>
```

SLAX Syntax

```
version 1.2;
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
     ...
     var $result = jcs:invoke('get-software-information');
     ...
```

The following example maps the `slax` prefix to the namespace identified by the URI `http://xml.libslax.org/slax`. The script then calls the `slax:get-input()` function with one string argument.

SLAX Syntax

```
version 1.2;
ns slax = "http://xml.libslax.org/slax";
    ...
```

```
    var $input = slax:get-input($prompt);
    ...
```

Using Extension Functions in Python Scripts

Python automation scripts that import the `jcs` module can use a Python version of supported Junos OS and SLAX extension functions. To determine which extension functions are supported in Python scripts, see "Understanding Extension Functions in Junos OS Automation Scripts" on page 337.

To call the equivalent extension function in a Python script, first include the `import jcs` statement and any required variable declarations. Then call the function using `jcs.`*`function_name`*`()`, and pass along any required or optional arguments. Note that in Python scripts, the extension function names must use underscores instead of hyphens. For example:

Python Syntax

```
import jcs

if __name__ == '__main__':
    name = jcs.get_input("Enter name: ")
    jcs.output(name)
```

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
Global Parameters and Variables in Junos OS Automation Scripts | 327
Use Named Templates in Junos OS Automation Scripts | 428

Using the sysctl() Extension Function on Junos Devices

Junos OS and Junos OS Evolved run on *nix-like operating systems, which enable you to retrieve various kernel state and process information. You can invoke the `sysctl()` extension function in your automation scripts to retrieve this same kernel state information, similar to how you would use the `sysctl` command in the shell to retrieve these values. The `sysctl()` function takes the same variable names that you would provide for the `sysctl` command in the shell. The variable name is a MIB-style name, which uses a dotted set of components. Because Junos OS is based on FreeBSD and Junos OS Evolved runs natively on Linux, the sysctl variables and variable names are different for each OS.

NOTE: You can execute the `sysctl -a` command in the shell to see the entire list of available states and the corresponding names that you can provide as arguments to the `sysctl()` function. However, the output can be extensive.

For example, on Junos OS, the following sample SLAX op script retrieves and prints the values for the sysctl states `kern.hostname` and `hw.product.model`:

```
version 1.1;
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> {
      var $host = jcs:sysctl("kern.hostname");
      expr jcs:output($host);
      var $model = jcs:sysctl("hw.product.model");
      expr jcs:output($model);
    }
}
```

```
user@router1> op sysctl-junos
router1
mx960
```

Similarly, on Junos OS Evolved, the following sample SLAX op script retrieves and prints the values for the sysctl states `kernel.hostname` and `kernel.osrelease`:

```
version 1.1;
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> {
      var $host = jcs:sysctl("kernel.hostname");
      expr jcs:output($host);
      var $osrelease = jcs:sysctl("kernel.osrelease");
      expr jcs:output($osrelease);
    }
}
```

```
user@router2-re0> op sysctl-evo
router2-re0
5.2.60-yocto-standard-g9a086a2b7
```

NOTE: For Junos OS Evolved, the return type is always a string ("s"). If you omit the type argument, the default is "s".

Junos OS and Junos OS Evolved have different `sysctl` state names because the underlying operating systems are different. For example, in Junos OS, you can retrieve the kernel state for `hw.product.model`, but Junos OS Evolved does not have a similar `sysctl` variable name. In those cases, you can use RPCs or other means to retrieve the same information.

For example, the following SLAX op script executes the `get-software-information` RPC to retrieve the model name on Junos OS Evolved, which is equivalent to returning the `sysctl hw.product.model` value on Junos OS.

```
version 1.1;
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> {
      var $local = jcs:open();
      var $rpc = "get-software-information";
      var $result = jcs:execute($local, $rpc);
      expr jcs:output($result/product-model);
```

```
    }
}
```

```
user@router2-re0> op sysctl-evo-model
ptx10008
```

If you invoke the `sysctl()` function in a script and specify a nonexistent sysctl variable name, Junos OS does not generate an error, but Junos OS Evolved generates a `sysctl error: No such file or directory` error.

For example, suppose the `sysctl()` function requests the value for `hw.product.model`, which is a valid sysctl variable name on Junos OS but is not a valid sysctl variable name on Junos OS Evolved. If you execute the script on a device running Junos OS Evolved, the script emits the following output:

```
user@router2-re0> op sysctl-evo-invalid
error: sysctl error: No such file or directory
error: xmlXPathCompiledEval: No result on the stack.
error: runtime error: file /var/db/scripts/op/sysctl-evo-invalid.slax line 10 element variable
error: Failed to evaluate the expression of variable 'model'.
```

RELATED DOCUMENTATION

sysctl() Function (Python, SLAX, and XSLT) | 418

CHAPTER 12

Extension Functions in the jcs and slax Namespaces

IN THIS CHAPTER

base64-decode() Function (SLAX)

IN THIS SECTION

Namespaces

```
http://xml.libslax.org/slax
```

SLAX Syntax

```
string slax:base64-encode(string, <control-string>)
```

Description

Decode BASE64 encoded data. BASE64 is a means of encoding arbitrary data into a radix-64 format that is more easily transmitted, typically using STMP or HTTP.

Include the optional control string argument to replace any non-XML control characters in the decoded string with the specified string. If the argument is an empty string, non-XML characters are removed. The decoded data is returned to the caller.

Parameters

control-string	(Optional) String to replace non-XML control characters in the decoded string. Use an empty string argument to remove the non-XML characters.
string	BASE64 encoded data.

Return Value

string	Decoded data.

Usage Examples

```
var $real-data = slax:base64-decode($encoded-data, "@");
```

Release Information

Function introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
base64-encode() Function (SLAX) | 352

base64-encode() Function (SLAX)

IN THIS SECTION

Namespaces

```
http://xml.libslax.org/slax
```

SLAX Syntax

```
string slax:base64-encode(string)
```

Description

Encode a string of data in the BASE64 encoding format. BASE64 is a means of encoding arbitrary data into a radix-64 format that is more easily transmitted, typically using STMP or HTTP.

Parameters

string	Input data string.

Return Value

string	Encoded data.

Usage Examples

```
var $encoded-data = slax:base64-encode($real-data);
```

Release Information

Function introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
base64-decode() Function (SLAX) | 350

break-lines() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

SLAX Syntax

```
var $lines = prefix:break-lines(expression);
```

XSLT Syntax

```
<xsl:variable name="lines" select="prefix: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.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

Parameters

expression	Original output.

Return Value

$lines	Output broken up into lines.

Usage Examples

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

Release Information

Function introduced in Junos OS Release 7.6

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337

parse-ip() Function (SLAX and XSLT) and parse_ip() (Python) | 402

regex() Function (SLAX and XSLT) | 409

split() Function (SLAX and XSLT) | 415

close() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
```

SLAX Syntax

```
var $results = jcs:close(connection);
```

XSLT Syntax

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

Description

Close a previously opened connection handle.

Parameters

connection 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);
```

Release Information

Function introduced in Junos OS Release 9.3.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
execute() Function (SLAX and XSLT) | 370
open() Function (SLAX and XSLT) | 394

dampen() Function (Python, SLAX, and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

Python Syntax

```
result = jcs.dampen(tag-string, max, interval)
```

SLAX Syntax

```
var $result = prefix:dampen(tag-string, max, interval);
```

XSLT Syntax

```
<xsl:variable name="result" select="prefix: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.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

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 */
}
```

Release Information

Function introduced in Junos OS Release 9.4.

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

document() Function (SLAX)

IN THIS SECTION

Namespaces

```
http://xml.libslax.org/slax
```

SLAX Syntax

```
string slax:document(url, <options>)
```

Description

Read data from a file or URL. Optional arguments specify the character encoding scheme and the encoding format and define the replacement string for non-XML control characters.

The `slax:document()` function reads text from the specified file and returns a string containing that text. The data can be encoded in any character set and can be BASE64 encoded. The default character set is "utf-8". You can set the `format` option to "base64" to decode base64-encoded files.

After you read the data, you can convert the data in the string into a more useful format. For example, you can use the `xutil:string-to-xml()` or `xutil:json-to-xml()` libslax function to convert the data in the string into the native representation of that data in XML. For more information about the libslax `xutil` library, see "libslax Default Extension Libraries: bit, curl, db, os, and xutil" on page 224.

XML cannot represent control characters. If a file contains control characters, the `slax:document()` function removes the control characters by default. Automatically removing the control characters enables you to convert the data in the string to XML. Alternatively, instead of removing the control characters, you can include the `non-xml` option to replace the characters with a single string.

> **NOTE**: Calling the `slax:document()` function to read the same file multiple times within a script does not reflect any changes made to that file in the interim, because the file is cached when first read, and the original contents are returned every time the `slax:document()` function retrieves the same file.

Parameters

`options` (Optional) Specify the character encoding scheme and format of the data, and define the replacement string for non-XML control characters.

Option	Description
`<encoding>` *string*	Character encoding scheme. For example "ascii" or "utf-8".
`<format>` *string*	"base64" for BASE64-encoded data.
`<non-xml>` *string*	String used to replace non-XML control characters. If the value is an empty string, non-XML characters are removed.

url File or URL from which to read data.

Return Value

string String representing the data.

Usage Examples

```
var $data = slax:document($url);

var $options := {
    <encoding> "ascii";
    <format> "base64";
    <non-xml> "#";
```

```
}
var $data2 = slax:document($url, $options);
```

Release Information

Function introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

emit_error() Function (Python)

IN THIS SECTION

Syntax

Python Syntax

```
jcs.emit_error(string)
```

Description

Generate an error message on the console from a Python automation script. The error message is preceded by the text `error:` .

When used in commit scripts, this function also causes the commit operation to fail.

> **NOTE**: If the `jcs.emit_error` string argument includes a null character sequence (`\0`), the script generates an `Invalid number of arguments` error. Additionally, if the string argument consists exclusively of an empty string or escape sequences (`\n \t \b \v \f \r`) and spaces, Junos OS does not emit any message.

Parameters

string	String describing the error.

Usage Examples

```
import jcs

def main():
    ...
    jcs.emit_error("Error message from a Python automation script")

if __name__ == '__main__':
    main()
```

Release Information

Function introduced in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Generate a Custom Warning, Error, or System Log Message in Commit Scripts | 572
Example: Generate a Custom Error Message | 586
emit_warning() Function (Python) | 365

emit_snmp_attributes Function (Python)

IN THIS SECTION

Syntax

Python Syntax

```
jcs.emit_snmp_attributes(snmp_oid, snmp_oid_type, snmp_oid_value)
```

Description

Return the attributes for the requested MIB object from a Python SNMP script.

SNMP scripts provide the flexibility to support custom MIBs. SNMP scripts are triggered automatically when the SNMP manager requests information for an unsupported object identifier (OID) that is mapped to an SNMP script. When the script is invoked, the `jcs.emit_snmp_attributes()` function emits the data for the requested object. The script acts like an SNMP subagent, and the system sends the return value from the script to the network management system (NMS).

Parameters

snmp_oid String containing the OID about which the script is returning information.

snmp_oid_type String that specifies the type of the data being returned for the requested MIB object. Acceptable values are `Counter32`, `Counter64`, `Integer32`, `Unsigned32`, and `Octet String`.

snmp_oid_value Data to return for the requested MIB object.

Usage Examples

The following example Python SNMP script processes unsupported object identifiers (OIDs) that are mapped to the script in the Junos OS configuration. The `emit_snmp_attributes()` function returns the attributes for the requested OID.

```
import jcs

def main():

    snmp_action = jcs.get_snmp_action()
    snmp_oid = jcs.get_snmp_oid()

    jcs.syslog("8", "snmp_action = ", snmp_action, " snmp_oid = ", snmp_oid)

    if snmp_action == 'get':
        if snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.1':
            jcs.emit_snmp_attributes(snmp_oid, "Integer32", "211")
        elif snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.2':
            jcs.emit_snmp_attributes(snmp_oid, "Integer32", "429")

    elif snmp_action == 'get-next':
        if snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1':
            jcs.emit_snmp_attributes(".1.3.6.1.4.1.2636.13.61.1.9.1.1.1", "Integer32", "211")
        elif snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.1':
            jcs.emit_snmp_attributes(".1.3.6.1.4.1.2636.13.61.1.9.1.1.2", "Integer32", "429")

if __name__ == '__main__':
    main()
```

Release Information

Function introduced in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Example: Process Unsupported OIDs with an SNMP Script | 1133
get_snmp_action() Function (Python) | 386
get_snmp_oid() Function (Python) | 388

emit_warning() Function (Python)

IN THIS SECTION

Syntax

Python Syntax

```
jcs.emit_warning(string)
```

Description

Generate a warning message on the console from a Python automation script. The warning message is preceded by the text `warning:` .

NOTE: If the `jcs.emit_warning` string argument includes a null character sequence (`\0`), the script generates an `Invalid number of arguments` error. Additionally, if the string argument consists exclusively of an empty string or escape sequences (`\n \t \b \v \f \r`) and spaces, Junos OS does not emit any message.

Parameters

string	String describing the warning.

Usage Examples

```
import jcs

def main():
    ...
    jcs.emit_warning("Warning message from a Python automation script")

if __name__ == '__main__':
    main()
```

Release Information

Function introduced in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Generate a Custom Warning, Error, or System Log Message in Commit Scripts | 572
Example: Generate a Custom Warning Message | 580
emit_error() Function (Python) | 361

empty() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

SLAX Syntax

```
var $result = prefix:empty(node-set | string);
```

XSLT Syntax

```
<xsl:variable name="result" select="prefix: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.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

Parameters

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

Return Value

result	Boolean value, which is `true` if the argument is empty.

Usage Examples

In the following example, if `$set` is empty, the script executes 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";
}
```

Release Information

Function introduced in Junos OS Release 7.6

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
first-of() Function (SLAX and XSLT) | 372

evaluate() Function (SLAX)

IN THIS SECTION

Namespaces

```
http://xml.libslax.org/slax
```

SLAX Syntax

```
object slax:evaluate(expression);
```

Description

Evaluate a SLAX expression and return the results of the expression. This supports expressions using the extended syntax provided by SLAX in addition to what is allowed in XPath.

Parameters

expression	SLAX expression to evaluate.

Return Value

object	Result of the expression.

Usage Examples

```
var $result = slax:evaluate("expr[name == '&']");
```

Release Information

Function introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337

execute() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
```

SLAX Syntax

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

XSLT Syntax

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

Description

Execute a *remote procedure call* (RPC) within the context of a specific management session in SLAX and XSLT automation scripts. Any number of RPCs may be executed within the same session until the session is closed with the `jcs:close()` function. This is in contrast to the `jcs:invoke()` function, which also performs RPC calls, but uses a unique session per command.

NOTE: Python automation scripts can use Junos PyEZ APIs to execute RPCs on a local or remote device.

Parameters

connection Connection handle generated by a call to the `jcs:open()` function.

rpc Remote procedure call (RPC) to execute.

Return Value

result Results of the executed RPC, which include 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. By default, the results are in XML format equivalent to the output produced with the `| display xml` option in the CLI.

Usage Examples

In the following example, the `$rpc` variable is declared and initialized with the Junos XML `<get-interface-information>` element. A call to the `jcs:open()` function generates a connection handle to the remote device at IP address 198.51.100.1. The user's login and password are provided as arguments to `jcs:open()` to provide access to the remote device. The code calls `jcs:execute()` and passes in the connection handle and RPC as arguments. Junos OS on the remote device processes the RPC and returns the results, which are stored in the `$results` variable.

```
var $rpc = <get-interface-information>;
var $connection = jcs:open('198.51.100.1', 'bsmith', 'test123');
var $results = jcs:execute($connection, $rpc);
expr $results;
expr jcs:close($connection);
```

The equivalent XSLT code is:

```
<xsl:variable name="connection" select="jcs:open('198.51.100.1', 'bsmith', 'test123')"/>
<xsl:variable name="rpc">
   <get-interface-information/>
</xsl:variable>
<xsl:variable name="results" select="jcs:execute($connection, $rpc)"/>
<xsl:value-of select="$results"/>
<xsl:value-of select="jcs:close($connection)"/>
```

Release Information

Function introduced in Junos OS Release 9.3.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
close() Function (SLAX and XSLT) | 355
invoke() Function (SLAX and XSLT) | 392
open() Function (SLAX and XSLT) | 394

first-of() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

SLAX Syntax

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

XSLT Syntax

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

Description

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

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

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 declaration 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";
   }
}
```

See also "Example: Display DNS Hostname Information Using an Op Script" on page 892.

Release Information

Function introduced in Junos OS Release 7.6.

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

RELATED DOCUMENTATION

Example: Display DNS Hostname Information Using an Op Script | 892

Understanding Extension Functions in Junos OS Automation Scripts | 337

empty() Function (SLAX and XSLT) | 366

get-command() Function (SLAX)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

SLAX Syntax

```
string = prefix:get-command(string);
```

Description

Prompt the user for input and return the input as a string. If the readline (or libedit) library was found at install time, the return string is entered in the readline history, and will be available using the readline history keystrokes (Ctrl+P and Ctrl+N).

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

Parameters

string	Prompt text.

Return Value

string	Command text entered by the user.

Usage Examples

```
var $response = slax:get-command("# ");
```

Release Information

Function introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337

get-hello() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
```

SLAX Syntax

```
var $capabilities = jcs:get-hello(connection);
```

XSLT Syntax

```
<xsl:variable name="capabilities" select="jcs:get-hello(connection)"/>
```

Description

Return the session ID and the capabilities of the NETCONF server during a NETCONF session.

During session establishment, the NETCONF server and client application each emit a `<hello>` element to specify which operations, or *capabilities*, they support from among those defined in the NETCONF specification or published as proprietary extensions. The `<hello>` element encloses the `<capabilities>` element and the `<session-id>` element, which specifies the session ID for this NETCONF session.

Within the `<capabilities>` element, a `<capability>` element specifies each supported function. Each capability defined in the NETCONF specification is represented by a uniform resource name (URN).

Capabilities defined by individual vendors are represented by uniform resource identifiers (URIs), which can be URNs or URLs.

Parameters

connection Connection handle generated by a call to the `jcs:open()` function.

Return Value

capabilities XML node set that specifies which operations, or *capabilities*, the NETCONF server supports. The node set also includes the session ID.

Usage Examples

In the following code snippet, the user, bsmith, establishes a NETCONF session on the default port with the remote device, fivestar, which is running Junos OS. Since the code does not specify a value for the password, the user is prompted for a password during script execution. Once authentication is established, the code calls the `jcs:get-hello()` function and stores the return value in the variable `$hello`, which is then printed to the CLI.

```
var $netconf := {
     <method> "netconf";
     <username> "bsmith";
}
var $connection = jcs:open("fivestar", $netconf);
var $hello = jcs:get-hello($connection);
expr jcs:output($hello);
expr jcs:close($connection);
```

The CLI displays the following output:

```
bsmith@fivestar's password:

urn:ietf:params:xml:ns:netconf:base:1.0
urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
urn:ietf:params:xml:ns:netconf:capability:validate:1.0
urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
http://xml.juniper.net/netconf/junos/1.0
```

```
http://xml.juniper.net/dmi/system/1.0

20847
```

Release Information

Function introduced in Junos OS Release 11.4.

RELATED DOCUMENTATION

get-input() Function (SLAX and XSLT) and get_input() (Python)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

Python Syntax

```
input = jcs.get_input('string')
```

SLAX Syntax

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

XSLT Syntax

```
<xsl:variable name="input" select="prefix: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 commit or event scripts.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

NOTE: If the `jcs.get_input` string argument includes a null character sequence (`\0`), Python automation scripts generate an `Invalid number of arguments` error.

Parameters

string	CLI prompt text.

Return Value

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 SLAX 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: ");
```

In Python:

```
username = jcs.get_input("Enter login id: ")
```

Release Information

Function introduced in Junos OS Release 9.4.

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

get-protocol() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
```

SLAX Syntax

```
var $protocol = jcs:get-protocol(connection);
```

XSLT Syntax

```
<xsl:variable name="protocol" select="jcs:get-protocol(connection)"/>
```

Description

Return the session protocol associated with the connection handle. The protocol values are `junoscript`, `netconf`, and `junos-netconf`.

Parameters

`connection`	Connection handle generated by a call to the `jcs:open()` function.

Return Value

`protocol`	Session protocol associated with the connection handle. The values are `junoscript`, `netconf`, and `junos-netconf`.

Usage Examples

In the following code snippet, the user, bsmith, establishes a NETCONF session on the default port with the remote device, fivestar. Since the code does not specify a value for the password, the user is prompted for a password during script execution. Once authentication is established, the code calls the `jcs:get-protocol()` function and stores the return value in the variable `$protocol`, which is then printed to the CLI.

```
var $netconf := {
     <method> "netconf";
     <username> "bsmith";
}
var $connection = jcs:open("fivestar", $netconf);
var $protocol = jcs:get-protocol($connection);
expr jcs:output($protocol);
expr jcs:close($connection);
```

The CLI displays the following output:

```
bsmith@fivestar's password:

netconf
```

Release Information

Function introduced in Junos OS Release 11.4.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337

Understanding the Session Protocol in Automation Scripts | 477

Example: Specify the Session Protocol for a Connection within Scripts | 480

get-hello() Function (SLAX and XSLT) | 377

open() Function (SLAX and XSLT) | 394

get-secret() Function (SLAX and XSLT) and get_secret() (Python)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

Python Syntax

```
input = jcs.get_secret('string')
```

SLAX Syntax

```
var $input = prefix:get-secret(string);
```

XSLT Syntax

```
<xsl:variable name="input" select="prefix: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 commit or event scripts.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

Parameters

string	CLI prompt text.

Return Value

`input`	Text typed by the user and stored as a string.

Usage Examples

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

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

In Python:

```
password = jcs.get_secret("Enter password")
```

Release Information

Function introduced in Junos OS Release 9.5R2.

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
get-input() Function (SLAX and XSLT) and get_input() (Python) | 379
output() Function (Python, SLAX, and XSLT) | 399
printf() Function (Python, SLAX, and XSLT) | 405
progress() Function (Python, SLAX, and XSLT) | 407

get_snmp_action() Function (Python)

IN THIS SECTION

Syntax

Python Syntax

```
snmp_action = jcs.get_snmp_action()
```

Description

Retrieve the SNMP action passed to a Python SNMP script.

Return Value

`snmp_action`	SNMP action value passed to the script.

Usage Examples

The following example Python SNMP script processes unsupported object identifiers (OIDs) that are mapped to the script in the Junos OS configuration. The `get_snmp_action()` function returns the value of the action argument passed into the script.

```
import jcs

def main():

    snmp_action = jcs.get_snmp_action()
    snmp_oid = jcs.get_snmp_oid()

    jcs.syslog("8", "snmp_action = ", snmp_action, " snmp_oid = ", snmp_oid)

    if snmp_action == 'get':
        if snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.1':
            jcs.emit_snmp_attributes(snmp_oid, "Integer32", "211")
        elif snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.2':
            jcs.emit_snmp_attributes(snmp_oid, "Integer32", "429")

    elif snmp_action == 'get-next':
        if snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1':
            jcs.emit_snmp_attributes(".1.3.6.1.4.1.2636.13.61.1.9.1.1.1", "Integer32", "211")
        elif snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.1':
            jcs.emit_snmp_attributes(".1.3.6.1.4.1.2636.13.61.1.9.1.1.2", "Integer32", "429")

if __name__ == '__main__':
    main()
```

Release Information

Function introduced in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Example: Process Unsupported OIDs with an SNMP Script | 1133
emit_snmp_attributes Function (Python) | 363
get_snmp_oid() Function (Python) | 388

Understanding Extension Functions in Junos OS Automation Scripts | 337

get_snmp_oid() Function (Python)

IN THIS SECTION

Syntax

Python Syntax

```
snmp_oid = jcs.get_snmp_oid()
```

Description

Retrieve the SNMP object identifier (OID) passed to a Python SNMP script.

Return Value

`snmp_oid`	SNMP OID value to process.

Usage Examples

The following example Python SNMP script processes unsupported object identifiers (OIDs) that are mapped to the script in the Junos OS configuration. The `get_snmp_oid()` function returns the value of the OID argument passed into the script.

```
import jcs

def main():

    snmp_action = jcs.get_snmp_action()
    snmp_oid = jcs.get_snmp_oid()

    jcs.syslog("8", "snmp_action = ", snmp_action, " snmp_oid = ", snmp_oid)

    if snmp_action == 'get':
        if snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.1':
            jcs.emit_snmp_attributes(snmp_oid, "Integer32", "211")
        elif snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.2':
            jcs.emit_snmp_attributes(snmp_oid, "Integer32", "429")

    elif snmp_action == 'get-next':
        if snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1':
            jcs.emit_snmp_attributes(".1.3.6.1.4.1.2636.13.61.1.9.1.1.1", "Integer32", "211")
        elif snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.1':
            jcs.emit_snmp_attributes(".1.3.6.1.4.1.2636.13.61.1.9.1.1.2", "Integer32", "429")

if __name__ == '__main__':
    main()
```

Release Information

Function introduced in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Example: Process Unsupported OIDs with an SNMP Script | 1133
emit_snmp_attributes Function (Python) | 363
get_snmp_action() Function (Python) | 386

hostname() Function (Python, SLAX, and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
```

Python Syntax

```
name = jcs.hostname(address)
```

SLAX Syntax

```
var $name = jcs:hostname(address);
```

XSLT Syntax

```
<xsl:variable name="name" select="jcs:hostname(address)"/>
```

Description

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

Parameters

address	String containing an IPv4 or IPv6 address.

Return Value

`name`	Hostname associated with the IP address.

Usage Examples

The following SLAX example initializes the variable `address` with the IP address 198.51.100.1. The `$address` variable is passed as the argument to the `jcs:hostname()` function. If the DNS server is configured on the device, the function will resolve the IP address and return the fully qualified domain name, which is stored in the variable `host`.

```
var $address = "198.51.100.1";
var $host = jcs:hostname($address);
```

In XSLT:

```
<xsl:variable name="address" select="198.51.100.1">
<xsl:variable name="host" select="jcs:hostname($address)"/>
```

In Python:

```
host = jcs.hostname("198.51.100.1")
```

Release Information

Function introduced in Junos OS Release 7.6.

Support for SLAX syntax added in Junos OS Release 8.2.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

invoke() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
```

SLAX Syntax

```
var $result = jcs:invoke(rpc, "no-login-logout");
```

XSLT Syntax

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

Description

Invoke a *remote procedure call* (RPC) on the local device. The function requires one argument, either a string containing a Junos XML API RPC, or a tree containing an RPC. The result contains the contents of the `<rpc-reply>` element, not including the `<rpc-reply>` tag. An RPC allows you to perform functions equivalent to Junos OS operational mode commands.

NOTE: Python automation scripts can use Junos PyEZ APIs to execute RPCs on a local or remote device.

Parameters

rpc	String containing a Junos XML API RPC or a tree containing an RPC.
no-login-logout	(Optional) In SLAX commit and event scripts, suppress UI_LOGIN_EVENT and UI_LOGOUT_EVENT messages in system log files when the script logs in as root to execute the RPC.

Return Value

result	Results of the executed RPC, which include the contents of the `<rpc-reply>` element, but not the `<rpc-reply>` tag itself. By default, the results are in XML format equivalent to the output produced with the `| display xml` option in the CLI.

Usage Examples

The following example tests to see if the `interface` argument is included on the command line when the script is executed. If the argument is provided, the output of the `show interfaces terse` operational mode command is narrowed to include only information about the specified interface.

```
<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 the Junos XML API RPC `get-software-information`, and stores the unmodified output in the variable `sw`:

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

Release Information

Function introduced in Junos OS Release 7.6.

`no-login-logout` parameter added in Junos OS Release 21.1R1 for commit and event scripts.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337

execute() Function (SLAX and XSLT) | 370

hostname() Function (Python, SLAX, and XSLT) | 390

sleep() Function (SLAX and XSLT) | 413

open() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
```

SLAX Syntax

```
var $connection = jcs:open();
var $connection = jcs:open(remote-hostname, <username>, <passphrase>, <routing-instance-name>);
var $connection = jcs:open(remote-hostname, <session-options>);
```

XSLT Syntax

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

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.

NOTE: Devices running Junos OS Evolved support only password-less login when establishing a local or remote connection using the `jcs:open()` function. They do not support supplying a password as a function argument or using an interactive password as is supported by Junos OS. To effect a local or remote connection, execute this functionality by way of a password-less login or else authentication issues could be encountered.

To redirect the SSH connection to originate from within a specific routing instance, include the routing instance name in the connection parameters. The routing instance must be configured at the `[edit routing-instances]` hierarchy level, and the remote device must be reachable either using the routing table for that routing instance or from one of the interfaces configured under that routing instance.

Starting in Junos OS Release 11.4, the new parameter, *session-options*, supports the option to create a session either with the Junos XML protocol server on devices running Junos OS or with the NETCONF server on devices where NETCONF service over SSH is enabled. Previously, the function supported only sessions with the Junos XML protocol server on devices running Junos OS.

NOTE: Python automation scripts can use Junos PyEZ APIs to establish a session and execute RPCs on a local or remote device.

The connection handle is closed with the `jcs:close()` function.

Parameters

passphrase — (Optional) User's login passphrase. If you do not specify a passphrase and it is required for authentication, you should be prompted for one during script execution by the device to which you are connecting.

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. If you specify a session type, this parameter is required.

routing-instance-name — (Optional) Routing instance from within which the SSH connection originates.

session-options — (Optional) XML node set that specifies the session protocol and connection parameters. The structure of the node set is:

```
var $session-options := {
    <instance> "routing-instance-name";
    <method> ("junoscript" | "netconf" | "junos-netconf");
    <passphrase> "passphrase";
    <password> "password";
    <port> "port-number";
    <routing-instance> "routing-instance-name";
```

```
    <username> "username";
}
```

- <instance>—(Optional) Routing instance from within which the SSH connection originates. This element is identical to <routing-instance>.
- <method>—(Optional) Session protocol. The protocol is one of three values: junoscript, netconf, or junos-netconf. If you do not specify a protocol, a junoscript session is created by default. A <method> value of junoscript establishes a session with the Junos XML protocol server on a device running Junos OS. A <method> value of netconf establishes a session with a NETCONF server over an SSHv2 connection. A <method> value of junos-netconf establishes a session with a NETCONF server over an SSHv2 connection on a device running Junos OS.
- <passphrase> or <password>—(Optional) User's login passphrase. If you do not specify a passphrase and it is required for authentication, you should be prompted for one during script execution by the device to which you are connecting.
- <port>—(Optional) Server port number for netconf and junos-netconf sessions. For NETCONF sessions, jcs:open() connects to the NETCONF server at the default port 830. If you specify a value for <port>, jcs:open() connects to the given port instead. Specifying a port number has no impact on junoscript sessions, which are always established over SSH port 22.
- <routing-instance>—(Optional) Routing instance from within which the SSH connection originates. This element is identical to <instance>.
- <username>—(Optional) User's login name. If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script.

username (Optional) User's login name. If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script.

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 the passphrase “test123” obtains a connection handle to the remote device, fivestar:

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

The following example shows how the user, bsmith, with the passphrase “test123” creates a `junos-netconf` session with a device running Junos OS:

```
var $options := {
     <method> "junos-netconf";
     <username> "bsmith";
     <passphrase> "test123";
}
var $connection = jcs:open("fivestar", $options);
```

Release Information

Function introduced in Junos OS Release 9.3.

Support for NETCONF sessions added in Junos OS Release 11.4.

Support for routing instances added in Junos OS Release 12.2.

Support for Junos OS Evolved added in Junos OS Evolved Release 18.3R1.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
Understanding the Session Protocol in Automation Scripts | 477

output() Function (Python, SLAX, and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

Python Syntax

```
jcs.output(string, <string>)
```

SLAX Syntax

```
expr prefix:output(string, <string>);
```

XSLT Syntax

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

Description

Display one or more lines of output text, either on the CLI (when used in op scripts), or to the output file (when used in event scripts). The function can be called with either a single string argument or with multiple string arguments. Multiple arguments are concatenated into a single string. A newline terminates the output text.

The `output()` function is not supported in commit scripts. SLAX and XSLT commit scripts use the `<xnm:warning>` and `<xnm:error>` result tree elements to display text on the CLI, and Python commit scripts use the `emit_warning()` and `emit_error()` functions.

The `output()` function displays the text immediately rather than waiting until the conclusion of the script, which differs from the SLAX and XSLT `<output>` element. This makes it suitable for scripts where user interaction is required or when status messages should be displayed during script processing. In SLAX and XSLT scripts, `jcs:output()` returns an empty node set, which can be ignored. Therefore, the `jcs:output()` function is normally called with the `expr` statement, rather than assigning its result to a variable.

The following escape characters are supported in the output text:

- \\ –Backslash (as of Junos OS Release 10.2)
- \r –Carriage Return
- \" –Double quote (as of Junos OS Release 10.1R2)
- \n –Newline
- \' – Single quote
- \t –Tab

Starting in Junos OS Release 10.2, the maximum length for output text is 10 KB, and longer strings are truncated to the supported length.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

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.')"/>
```

Python syntax:

```
jcs.output('The VPN is up.')
```

Release Information

Function introduced in Junos OS Release 7.6.

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
get-input() Function (SLAX and XSLT) and get_input() (Python) | 379
get-secret() Function (SLAX and XSLT) and get_secret() (Python) | 384
printf() Function (Python, SLAX, and XSLT) | 405
progress() Function (Python, SLAX, and XSLT) | 407
syslog() Function (Python, SLAX, and XSLT) | 420
trace() Function (Python, SLAX, and XSLT) | 424

parse-ip() Function (SLAX and XSLT) and parse_ip() (Python)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
```

Python Syntax

```
result = jcs.parse_ip("ipaddress/(prefix-length | netmask)")
```

SLAX Syntax

```
var $result = jcs:parse-ip("ipaddress/(prefix-length | netmask)");
```

XSLT Syntax

```
<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.

netmask Netmask defining the network portion of the address.

prefix-length Prefix length defining the number of bits used in the network portion of the address.

Return Value

result An array containing the following information. In SLAX and XSLT scripts, the array index starts at 1. In Python scripts, the array index begins at 0.

- 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 addresses (left blank for IPv6 addresses)

Usage Examples

The following two SLAX examples parse an IPv4 address and an IPv6 address and detail the resulting output:

```
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("2001:DB8::c50:8a:800:200C:417A/32");
```

- `$addr[1]` contains the host address `2001:db8:0:c50:8a:800:200c:417a`.
- `$addr[2]` contains the protocol family `inet6`.
- `$addr[3]` contains the prefix length `32`.
- `$addr[4]` contains the network address `2001:db8::`.
- `$addr[5]` is blank for IPv6 (`""`).

The following Python statement parses an IPv4 address. The values in the `addr` array are shown. Note that the array index begins at 0.

```
addr = jcs.parse_ip('10.1.2.10/255.255.255.0')
```

- `$addr[0]` contains the host address `10.1.2.10`.
- `$addr[1]` contains the protocol family `inet`.
- `$addr[2]` contains the prefix length `24`.
- `$addr[3]` contains the network address `10.1.2.0`.
- `$addr[4]` contains the netmask for IPv4 `255.255.255.0`.

Release Information

Function introduced in Junos OS Release 9.0.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
break-lines() Function (SLAX and XSLT) | 353
hostname() Function (Python, SLAX, and XSLT) | 390
regex() Function (SLAX and XSLT) | 409
split() Function (SLAX and XSLT) | 415

printf() Function (Python, SLAX, and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

Python Syntax

```
jcs.printf(expression)
```

SLAX Syntax

```
expr prefix:printf(expression);
```

XSLT Syntax

```
<xsl:value-of select="prefix: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 that emits the field only if it changed from the last time the function was called. This assumes that the expression's format string is unchanged.
- `jc`—Operator that capitalizes the first letter of the associated output string.
- `jt{TAG}`—Operator that emits the tag if the associated argument is not empty.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

Parameters

expression Format string containing an arbitrary number of format specifiers and associated arguments to output.

Usage Examples

In the following example, the `j1` operator suppresses printing the interface identifier so-0/0/0 in the second line of output, because the identifier argument has not changed from the first printing. The `jc` operator capitalizes the output strings `up` and `down`. The `jt{--}` operator does not print the `{--}` tag in the first line of output, because the associated output argument is an empty string. However, the tag is printed in the second line because the associated output is the non-empty string `test`.

```
<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', 'test')"/>
```

produces the following output:

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

Release Information

Function introduced in Junos OS Release 7.6.

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

progress() Function (Python, SLAX, and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

Python Syntax

```
jcs.progress(string)
```

SLAX Syntax

```
expr prefix:progress(string);
```

XSLT Syntax

```
<xsl:value-of select="prefix: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.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

Parameters

string	Text that is output to CLI session.

Usage Examples

SLAX syntax:

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

XSLT syntax:

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

Python syntax:

```
 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'
```

Release Information

Function introduced in Junos OS Release 7.6.

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337

get-input() Function (SLAX and XSLT) and get_input() (Python) | 379

get-secret() Function (SLAX and XSLT) and get_secret() (Python) | 384

output() Function (Python, SLAX, and XSLT) | 399

printf() Function (Python, SLAX, and XSLT) | 405

syslog() Function (Python, SLAX, and XSLT) | 420

trace() Function (Python, SLAX, and XSLT) | 424

regex() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

SLAX Syntax

```
var $result = prefix:regex(pattern, string);
```

XSLT Syntax

```
<xsl:variable name="result" select="prefix: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.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

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 regex 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 of the regex pattern to each of the specified strings.

```
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]
```

Release Information

Function introduced in Junos OS Release 7.6

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
break-lines() Function (SLAX and XSLT) | 353
parse-ip() Function (SLAX and XSLT) and parse_ip() (Python) | 402
split() Function (SLAX and XSLT) | 415

set_routing_instance() Function (Python)

IN THIS SECTION

Syntax

```
jcs.set_routing_instance(routing-instance-name)
```

Description

Use to program the protocol software (TCP/UDP) to use non-default routing instances. You can use this function in op scripts and on-box Juniper Extension Toolkit (JET) applications.

NOTE: In Junos OS Release 19.3R1, this feature is supported only on 32-bit architecture. Starting in Junos OS Release 19.4R1, this feature is also supported on 64-bit architecture.

Parameters

routing-instance-name String specifying the routing instance through which the connection is made.

Usage Examples

The following Python script uses the `jcs.set_routing_instance()` function to connect to a device through a non-default routing instance. In the script, you must set the non-default routing instance before you connect to the device.

Python script:

```
user@r1> file show /var/db/scripts/op/python-routing-instance.py
from junos import Junos_Context
from jnpr.junos import Device
from pprint import pprint
import jcs

user = Junos_Context['user-context']['user']
password = jcs.get_secret('Enter user password: ')
routing_instance = jcs.get_input('Enter routing instance: ')

# set routing instance option before connecting
jcs.set_routing_instance(routing_instance)
try:
    with Device(host='198.51.100.2', user=user, password=password) as dev:
        pprint (dev.facts)
except Exception as err:
    print (err)
```

Release Information

Function introduced in Junos OS Release 19.3R1 on MX Series routers.

sleep() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

SLAX Syntax

```
expr prefix:sleep(seconds, <milliseconds>);
```

XSLT Syntax

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

Description

Cause the script to pause 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.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

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)"/>
```

Release Information

Function introduced in Junos OS Release 7.6.

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
dampen() Function (Python, SLAX, and XSLT) | 356
hostname() Function (Python, SLAX, and XSLT) | 390
sysctl() Function (Python, SLAX, and XSLT) | 418

split() Function (SLAX and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

SLAX Syntax

```
var $substrings = prefix:split(expression, string, <limit>);
```

XSLT Syntax

```
<xsl:variable name="substrings" select="prefix: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.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

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. If *limit* is not specified, the result array size is equal to the number of substrings extracted from the original string as determined by the specified delimiter.

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"
```

Release Information

Function introduced in Junos OS Release 8.4

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

RELATED DOCUMENTATION

sysctl() Function (Python, SLAX, and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

Python Syntax

```
value = jcs.sysctl("sysctl-value", "(i | s)")
```

SLAX Syntax

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

XSLT Syntax

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

Description

Return the given `sysctl` value.

Python scripts require you to specify the return value's type. However, the type argument is optional in SLAX and XSLT scripts. In Junos OS, use '`i`' to specify an integer, and use '`s`' to specify a string. In Junos OS Evolved, you should only use '`s`'.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

Parameters

sysctl-value	Name of the `sysctl` value to return, for example, `kern.hostname` on Junos OS or `kernel.hostname` on Junos OS Evolved.

Return Value

`value`	Returned string or integer value.

Usage Examples

jcs:sysctl() Function (Junos OS)

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

jcs:sysctl() Function (Junos OS Evolved)

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

Release Information

Function introduced in Junos OS Release 7.6

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Using the sysctl() Extension Function on Junos Devices | 345
Understanding Extension Functions in Junos OS Automation Scripts | 337

syslog() Function (Python, SLAX, and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

Python Syntax

```
jcs.syslog(priority, message, <message2>)
```

SLAX Syntax

```
expr prefix:syslog(priority, message, <message2>);
```

XSLT Syntax

```
<xsl:value-of select="prefix:syslog(priority, messsage, <message2>)"/>
```

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 that is written to the system log file. Optionally, additional strings can be included in the argument list. The *message* argument is concatenated with any additional arguments, 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 configuration.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

Parameters

message String that is output to the system log file.

message2 (Optional) Any additional number of strings passed as arguments to the function. These are concatenated with the *message* argument and output to the system log file.

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. In Python scripts, the priority integer value must be passed in as a string.

Table 37 on page 422 and Table 38 on page 422 show the available facility and severity strings and their corresponding numeric values. The integer value of the *priority* parameter is calculated by multiplying the numeric value of the facility string by 8 and adding the numeric value of the severity string. For example, if the *facility.severity* string pair is `"pfe.alert"`, the priority value is 161 ((20 x 8)+1).

Table 37: 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
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 38: Severity Strings

Severity String	Description	Numeric Value
alert	Conditions that should be corrected immediately	1
crit	Critical conditions	2
debug	Debug messages	7

Table 38: Severity Strings *(Continued)*

Severity String	Description	Numeric Value
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 SLAX 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 arguments. For this case, the concatenated string "`mymessage mymessage2`" is output to the system log file.

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

Similarly, in Python:

```
jcs.syslog("pfe.alert", "message")
jcs.syslog("161", "message")
```

Release Information

Function introduced in Junos OS Release 7.6

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

trace() Function (Python, SLAX, and XSLT)

IN THIS SECTION

Namespaces

```
http://xml.juniper.net/junos/commit-scripts/1.0
http://xml.libslax.org/slax
```

Python Syntax

```
jcs.trace(string)
```

SLAX Syntax

```
expr prefix:trace(string);
```

XSLT Syntax

```
<xsl:value-of select="prefix:trace(string)"/>
```

Description

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

You must configure `traceoptions` under the respective script type in the configuration hierarchy in order to output a message to the trace file using the `trace()` function. The output goes to the configured trace file. If `traceoptions` is enabled, but no trace file is explicitly configured, the output goes to the default trace file for that script type.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

Parameters

string	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')"/>
```

Python syntax:

```
jcs.trace('test')
```

Release Information

Function introduced in Junos OS Release 7.6.

Support for the slax namespace http://xml.libslax.org/slax added in Junos OS Release 12.2.

Support for Python added in Junos OS Release 16.1R1.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
get-input() Function (SLAX and XSLT) and get_input() (Python) | 379
get-secret() Function (SLAX and XSLT) and get_secret() (Python) | 384
output() Function (Python, SLAX, and XSLT) | 399
printf() Function (Python, SLAX, and XSLT) | 405
progress() Function (Python, SLAX, and XSLT) | 407
syslog() Function (Python, SLAX, and XSLT) | 420

CHAPTER 13

Named Templates for Automation Scripting

IN THIS CHAPTER

Understanding Named Templates in Junos OS Automation Scripts

Junos OS provides several named templates that can be used in commit, op, event, and SNMP scripts to more easily accomplish scripting tasks on devices running Junos OS. The named templates reside in import files, which are included with the standard Junos OS installation available on switches, routers, and security devices running Junos OS. Table 39 on page 427 summarizes the Junos OS named templates.

For information about using named templates in automation scripts, see "Using Named Templates in Junos OS Automation Scripts" on page 428.

Table 39: Junos OS Named Templates

Template	Description	Supported in Python Scripts
`jcs:edit-path`	Generate an `<edit-path>` element suitable for inclusion in an `<xnm:error>` or `<xnm:warning>` element.	–
`jcs:emit-change` (XSLT/SLAX) `jcs.emit_change` (Python)	Generate a persistent or *transient change* to the configuration.	Y
`jcs:emit-comment`	Emit a simple comment that indicates a change was made by a *commit script*.	–

Table 39: Junos OS Named Templates *(Continued)*

Template	Description	Supported in Python Scripts
jcs:grep	Search a file for all instances matching a specified regular expression and write the matching strings and corresponding lines to the result tree.	–
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

Use Named Templates in Junos OS Automation Scripts | 428
Understanding Extension Functions in Junos OS Automation Scripts | 337
Global Parameters and Variables in Junos OS Automation Scripts | 327

Use Named Templates in Junos OS Automation Scripts

IN THIS SECTION

Junos OS provides several named templates that can be used in commit, op, event, and SNMP scripts to more easily accomplish scripting tasks on devices running Junos OS. The following sections outline how to import and use the templates for different script languages:

Using Named Templates in SLAX and XSLT Scripts

To use the named templates within SLAX and XSLT scripts, the script must import the **junos.xsl** file and also declare the appropriate namespace *Uniform Resource Identifier* (URI) in the style sheet declaration. The Junos OS named templates are defined in the namespace with the associated URI `http://xml.juniper.net/junos/commit-scripts/1.0`.

Import the **junos.xsl** file into the script by including the `<xsl:import/>` tag element in XSLT scripts or the `import` statement in SLAX scripts and specifying the **junos.xsl** file location.

SLAX and XSLT scripts generally map the `jcs` prefix to the URI to avoid name conflicts with standard XSLT or user-defined templates. The scripts then qualify the named templates with the appropriate prefix, which is expanded into its associated URI reference during processing. Map the `jcs` prefix to the URI by including the `xmlns:jcs` attribute in the opening `<xsl:stylesheet>` tag element for XSLT scripts or by including the `ns jcs` statement in SLAX scripts.

To call a named template in a script, include the `<xsl:call-template name="`*`template-name`*`">` element in XSLT scripts or the `call` statement in SLAX scripts and pass along any required or optional parameters. Template parameters are assigned by name and can appear in any order. This differs from functions where the arguments must be passed into the function in the precise order specified by the function definition.

The following example imports the **junos.xsl** file into a script and maps the `jcs` prefix to the namespace identified by the URI http://xml.juniper.net/junos/commit-scripts/1.0. The script demonstrates a call to the `jcs:edit-path` template.

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:for-each select="interfaces/interface[starts-with(name, 'so-')]">
        <xnm:warning>
            <xsl:call-template name="jcs:edit-path"/>
            <message>interface configured</message>
        </xnm:warning>
    </xsl:for-each>
    ...
</xsl: stylesheet>
```

SLAX Syntax

```
version 1.2;
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";
    ...
    for-each ( interfaces/interface[starts-with(name,'so-') ) {
        <xnm:warning> {
            call jcs:edit-path();
            <message> “interface configured”;
        }
    }
    ...
```

Using Named Template Functionality in Python Scripts

Python scripts that import the `jcs` module can execute functions that are the Python equivalent of the named templates used in SLAX and XSLT scripts. The functions provide essentially the same functionality in the Python script as the corresponding named template does in a SLAX or XSLT script. To determine which named templates have functionality that is supported in Python scripts, see "Understanding Named Templates in Junos OS Automation Scripts" on page 427.

To call the function equivalent to a named template within a Python script, include any required variable declarations, call the function, and pass along any required or optional arguments. Note that in Python scripts, the function names must use underscores instead of hyphens. For example:

Python Syntax

```
import jcs

if __name__ == '__main__':
    script = "system-check.py"
    change_xml = """<system><scripts><op>
                    <file><name>{0}</name></file></op>
                    </scripts></system>""".format(script)
    jcs.emit_change(change_xml, "change", "xml")
```

RELATED DOCUMENTATION

Understanding Named Templates in Junos OS Automation Scripts | 427

CHAPTER 14

Named Templates in the jcs Namespace

IN THIS CHAPTER

jcs:edit-path Template

IN THIS SECTION

XSLT Syntax

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

SLAX Syntax

```
call jcs:edit-path($dot=expression);
```

Description

Generate an <edit-path> element suitable for inclusion in an <xnm:error> or <xnm:warning> element. This template converts a location in the configuration hierarchy into the standard text representation that you would see in the Junos OS configuration mode banner. By default, the location of the configuration error is passed into the jcs:edit-path template as the value of dot. This location defaults to " . ", the current position in the XML hierarchy. You can alter the default by including a valid *XPath* expression for the dot parameter when you call the template.

Parameters

dot XPath expression specifying the hierarchy level. The default location is the position in the XML hierarchy that the script is currently evaluating. You can alter the default when you call the template by including a valid XPath expression either for the dot parameter in SLAX scripts or for the select attribute of the dot parameter in XSLT scripts.

Usage Examples

The following example demonstrates how to call the jcs:edit-path 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 code generates an `<xnm:warning>` element, which 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

Understanding Named Templates in Junos OS Automation Scripts | 427
call | 138
xnm:error (Junos XML) | 800
xnm:warning (Junos XML) | 802
xsl:call-template | 56

emit-change Template (SLAX and XSLT) and emit_change (Python)

IN THIS SECTION

Syntax

Python Syntax

```
jcs.emit_change(content, tag, format)
```

SLAX Syntax

```
call jcs:emit-change($dot=expression, $name = name($dot), $tag = "(change | transient-change)" {
    with $content = {
        ...
     }
    with $message = {
        expr "message";
     }
}
```

XSLT 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>message</xsl:text>
    </xsl:with-param>
    <xsl:with-param name="name" select="name($dot)"/>
    <xsl:with-param name="tag" select="'(change | transient-change)'"/>
</xsl:call-template>
```

Description

Generate a persistent or transient change to the configuration.

Parameters

content Content of the persistent or transient change. In SLAX and XSLT scripts, this is relative to `dot`. Python scripts must include the full configuration path representing all levels of the configuration hierarchy formatted as an XML string.

dot *XPath* expression specifying the hierarchy level at which the change will be made. The default location is the position in the XML hierarchy that the script is currently evaluating. You can alter the default when you call the template by including a valid XPath expression either for the `dot` parameter in SLAX scripts or for the `select` attribute of the `dot` parameter in *XSLT* scripts.

format Format of the configuration data loaded through a Python commit script. The only supported format is `xml`.

message Warning message displayed in 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, specify a valid XPath expression either for the `dot` parameter in SLAX scripts or for the `select` attribute of the `dot` parameter in XSLT scripts.

name 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 value `name($dot)`, which is the name of the element in `dot` (which in turn defaults to " . ", which is the current element).

tag Type of change to generate. Specify `'change'` to generate a *persistent change*, or specify `'transient-change'` to generate a *transient change*. If you omit this parameter in SLAX and XSLT scripts, the `jcs:emit-change` template defaults to generating a persistent change.

Usage Examples

The following example demonstrates how to call the `jcs:emit-change` template in an XSLT *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>
```

In SLAX and XSLT scripts, 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"/>
            ...
```

The following Python commit script generates a persistent change to the configuration:

```
import jcs

if __name__ == '__main__':
    script = "system-check.py"
    change_xml = """<system><scripts><op>
                    <file><name>{0}</name></file></op>
                    </scripts></system>""".format(script)
    jcs.emit_change(change_xml, "change", "xml")
```

RELATED DOCUMENTATION

Example: Impose a Minimum MTU Setting | 732

Understanding Named Templates in Junos OS Automation Scripts | 427

call | 138

xsl:call-template | 56

jcs:emit-comment Template

IN THIS SECTION

XSLT Syntax

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

Description

Generate a comment in the configuration from within a commit, event, or op 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

Understanding Named Templates in Junos OS Automation Scripts | 427
emit-change Template (SLAX and XSLT) and emit_change (Python) | 434

jcs:grep Template

IN THIS SECTION

XSLT Syntax

```
<xsl:call-template name="jcs:grep">
    <xsl:with-param name="filename" select="filename"/>
```

```
    <xsl:with-param name="pattern" select="pattern"/>
</xsl:call-template>
```

SLAX Syntax

```
call jcs:grep($filename=filename, $pattern=pattern);
```

Description

Search the given input file for all instances matching the specified regular expression and write the matching strings and corresponding lines to the result tree. The pattern is matched to each line of the file. The template does not support matching a pattern spanning multiple lines.

> **NOTE**: The `jcs:grep` template supports a maximum input file size of 9.5MB. If an input file exceeds this maximum, the script generates an error regarding `xmlSAX2Characters: huge text node` and `Extra content at the end of the document`.

If the regular expression contains a syntax error, the template generates an error for every line of the file. For each match, the template adds a `<match>` element, which contains `<input>` and `<output>` child tags, to the result tree. The template writes the matching string to the `<output>` element and writes the corresponding matching line to the `<input>` element.

```
<match> {
     <input>
     <output>
}
```

Starting in Junos OS Release 11.1, if an absolute path is not specified for the input file, the default path is relative to the user's home directory for op scripts, and it is relative to the **/var/tmp/** directory for commit scripts and for event scripts that are enabled at the `[edit event-options event-script]` hierarchy level. For event scripts that are enabled at the `[edit system scripts]` hierarchy level, the default path is relative to the top-level directory, /.

Parameters

`filename` Absolute or relative path and filename of the file to search.

Starting in Junos OS Release 11.1, if you do not specify an absolute path, the path is relative to the user's home directory for op scripts, and it is relative to the **/var/tmp/** directory for commit scripts and for event scripts that are enabled at the `[edit event-options event-script]` hierarchy level. For event scripts that are enabled at the `[edit system scripts]` hierarchy level, the default path is relative to the top-level directory, /.

`pattern` Regular expression to match in the file.

RELATED DOCUMENTATION

Example: Search Files Using an Op Script | 925

Understanding Named Templates in Junos OS Automation Scripts | 427

SLAX Operators | 120

regex() Function (SLAX and XSLT) | 409

jcs:load-configuration Template

IN THIS SECTION

SLAX Syntax

```
call jcs:load-configuration(
    $action="(merge | override | replace)",
    $commit-options=node-set,
    $configuration=configuration-data,
    $connection=connection-handle,
    $rescue="rescue",
    $rollback=number);
```

XSLT 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:with-param name="rescue" select="&quot;rescue&quot;"/>
    <xsl:with-param name="rollback" select="number"/>
</xsl:call-template>
```

Description

Make structured changes to the Junos OS configuration using an *op script* or event 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.

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 operation. 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.
- synchronize—Synchronize the commit on both Routing Engines.

`configuration` XML configuration changes. The configuration changes are incorporated into the candidate configuration as specified by the `action` parameter.

`connection` Connection handle generated by a call to the `jcs:open()` function.

`rescue` Load and commit the rescue configuration, if one exists. The only acceptable value for this parameter is `"rescue"`. The `rescue` parameter is available starting in Junos OS Release 20.1R1.

`rollback` Revert to a previously committed configuration. Specify the rollback number of the configuration. The `rollback` parameter is available starting in Junos OS Release 12.2.

RELATED DOCUMENTATION

Change the Configuration Using SLAX and XSLT Scripts | 857
Example: Change the Configuration Using SLAX and XSLT Op Scripts | 864
Understanding Named Templates in Junos OS Automation Scripts | 427

jcs:statement Template

IN THIS SECTION

XSLT Syntax

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

SLAX Syntax

```
call jcs:statement($dot=expression);
```

Description

Generate a `<statement>` element suitable for inclusion in an `<xnm:error>` or `<xnm:warning>` element. This location defaults to " . ", the current position in the XML hierarchy. If the error is not at the current position in the XML hierarchy, you can alter the default when you call the template by including a valid *XPath* expression either for the `dot` parameter in SLAX scripts or for the `select` attribute of the `dot` parameter in *XSLT* scripts.

Parameters

dot XPath expression specifying the hierarchy level. The default location is the position in the XML hierarchy that the script is currently evaluating. You can alter the default when you call the template by including a valid XPath expression either for the `dot` parameter in SLAX scripts or for the `select` attribute of the `dot` parameter in XSLT scripts.

Usage Examples

The following example demonstrates how to call the `jcs:statement` 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: Impose a Minimum MTU Setting" on page 732.

RELATED DOCUMENTATION

7

PART

Manage Automation Scripts

CHAPTER 15

Store and Enable Scripts

IN THIS CHAPTER

Store and Enable Junos Automation Scripts

To use a Junos automation script, you must copy the script to the Junos device and enable it in the configuration. , as described in the following steps. By default, scripts are stored in and executed from the **/var/db/scripts** directory on the device's hard disk under the subdirectory appropriate to the script type. You can also store scripts in flash memory in the **/config/scripts** directory under the subdirectory appropriate to the script type.

1. Create the script.
2. Copy the script to the appropriate directory on the device for that script type. Only users who belong to the Junos OS `super-user` login class can access and edit files in the script directories on a device running Junos OS.

Table 40: Script Locations

Script Type	Hard Disk Location	Flash Memory Location
Commit script	**/var/db/scripts/commit**	**/config/scripts/commit**
Event script	**/var/db/scripts/event**	**/config/scripts/event**
Op script	**/var/db/scripts/op**	**/config/scripts/op**
Snmp script	**/var/db/scripts/snmp**	**/config/scripts/snmp**

NOTE: If the device has dual Routing Engines and you want to enable the script to execute on both Routing Engines, you can copy the script to the appropriate directory on both Routing Engines, or you can issue the `commit synchronize scripts` command to synchronize the configuration and copy the scripts to the other Routing Engine as part of the commit operation.

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

3. For unsigned Python scripts, ensure that the scripts meet the following requirements:

   - File owner is either root or a user in the Junos OS `super-user` login class.
   - Only the file owner has write permission for the file.
   - The `language python` or `language python3` statement is configured at the `[edit system scripts]` hierarchy level.

```
[edit system scripts]
user@host# set language (python | python3)
```

NOTE: Starting in Junos OS Release 16.1R3, unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file. Prior to Junos OS Release 16.1R3, unsigned Python scripts must only be owned by the root user.

NOTE: To enable a user who does not belong to the file's user or group class to execute an unsigned Python automation script, the script's file permissions must include read permission for others.

4. Enable the script by including the `file` *`filename`* statement at the appropriate hierarchy level for that script type.

   - `commit script`—Include 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 557.
   - `op script`—Include 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 847.

- `event script`—Include the `file` *`filename`* statement at the `[edit event-options event-script]` hierarchy level. For instructions, see "Enabling and Executing Event Scripts" on page 1099.

- `snmp script`—Include the `file` *`filename`* statement at the `[edit system scripts snmp]` hierarchy level. For instructions, see "Enabling an SNMP Script" on page 1128.

5. If you store scripts in and load them from flash memory, include the `load-scripts-from-flash` statement at the `[edit system scripts]` hierarchy level. For detailed information about storing scripts in flash memory, see "Storing Scripts in Flash Memory" on page 451.

```
[edit]
user@host# set system scripts load-scripts-from-flash
```

6. For Python event and SNMP scripts, configure the user under whose access privileges the script executes.

   - For event scripts:

```
[edit event-options event-script]
user@host# set file filename python-script-user username
```

   - For SNMP scripts:

```
[edit system scripts snmp]
user@host# set file filename python-script-user username
```

NOTE: If you do not configure the `python-script-user` statement, then by default, Junos OS executes Python event and SNMP scripts under the access privileges of the generic, unprivileged user and group `nobody`. Interactive Python scripts, such as commit and op scripts, run with the access privileges of the user who executes the command or operation that invokes the script.

NOTE: Starting in Junos OS Release 16.1R3, you can execute unsigned Python commit, event, op, and SNMP scripts using the access privileges of authorized users. In Junos OS Release 16.1R2 and earlier releases, all unsigned Python automation scripts are executed using the access privileges of the user and group `nobody`.

7. Commit the configuration.

```
[edit]
user@host# commit
```

Newly enabled commit scripts execute during the current commit operation. After the commit operation completes, enabled event scripts are loaded into memory and can be executed by an event policy that is triggered in response to system log events. For more information, see "Executing Event Scripts in an Event Policy" on page 1021. After the commit operation completes, op scripts can be executed on the device. For more information, see "Executing an Op Script on the Local Device" on page 851.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
16.1R3	Starting in Junos OS Release 16.1R3, unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.
16.1R3	Starting in Junos OS Release 16.1R3, you can execute unsigned Python commit, event, op, and SNMP scripts using the access privileges of authorized users.

RELATED DOCUMENTATION

Store Scripts in Flash Memory | **451**

Store and Use Imported Scripts and Script Functionality | **453**

Control the Execution of Commit Scripts During Commit Operations | **557**

load-scripts-from-flash (Scripts)

Store Scripts in Flash Memory

By default, Junos OS automation scripts are stored in and executed from the **/var/db/scripts** directory on the device's hard disk under the subdirectory appropriate to the script type. Optionally, you can store automation scripts in and execute them from the device's flash memory. In the unlikely event of a hard disk failure, storing scripts in flash memory reduces the impact of the failure, because you can still access and execute the scripts. The flash memory partition is typically small and also stores the device's active configuration, the last three committed configurations, and the rescue configuration, if it exists. Before storing scripts in flash memory, ensure that there is sufficient space to do so.

To instruct a device running Junos OS to load scripts from flash storage, configure the `load-scripts-from-flash` statement at the `[edit system scripts]` hierarchy level:

```
[edit]
user@host# set system scripts load-scripts-from-flash
```

When you configure the `load-scripts-from-flash` statement, all commit, event, op, SNMP, and script library scripts are loaded from the **/config/scripts** directory in flash memory under the subdirectory appropriate to the script type. You must manually move scripts from the hard disk to flash memory. They are not moved automatically. Similarly, if you delete the `load-scripts-from-flash` statement from the configuration, you must manually copy the scripts from the flash memory to the hard disk to ensure that the current versions of the scripts are executed. Changing the scripts' physical location has no effect on their operation.

The system uses the **/var/run/scripts** directory as a means of finding the current set of scripts. This path uses a symbolic link to reference either the **/var/db/scripts** directory on the hard disk or the **/config/scripts** directory in flash memory according to the current setting of the `load-scripts-from-flash` statement. If you do not configure the `load-scripts-from-flash` statement, **/var/run/scripts** references the **/var/db/scripts** directory on the device's hard disk. If you configure the `load-scripts-from-flash` statement, **/var/run/scripts** references the **/config/scripts** directory in flash memory.

```
user@host> file list detail /var/run/scripts
lrwxr-xr-x  1 root  wheel  16 May 9  09:51 /var/run/scripts@ -> /var/db/scripts/
```

To view the scripts on the device, list the contents of **/var/run/scripts/*type***, where *type* is the subdirectory appropriate to the script type. In the following example, the `load-scripts-from-flash` statement is not configured. In this case, listing the files for **/var/run/scripts/commit** returns the files in the **/var/db/scripts/commit** directory.

```
user@host>file list /var/run/scripts/commit

/var/run/scripts/commit:
commit-changes-load-replace.slax
commit-protect.slax
```

```
user@host> file list /var/db/scripts/commit
```

```
/var/db/scripts/commit:
commit-changes-load-replace.slax
commit-protect.slax
```

```
user@host> file list /config/scripts/commit

/config/scripts/commit:
```

RELATED DOCUMENTATION

Store and Enable Junos Automation Scripts | 448
Store and Use Imported Scripts and Script Functionality | 453
load-scripts-from-flash (Scripts)

Store and Use Imported Scripts and Script Functionality

SUMMARY

Junos OS provides a dedicated script library directory for storing scripts that you want to import into your commit, event, op, and SNMP scripts.

Junos OS provides a dedicated directory for script libraries, where users can store scripts and script functionality that then can be imported into any commit, event, op, or SNMP script. Upon installation, Junos OS creates the **/var/db/scripts/lib/** directory. Junos OS will not overwrite or erase any files in an existing **lib/** directory upon installation or upgrade.

If you store scripts in and run them from flash memory, both the executed scripts and the imported scripts must be present on the flash drive. When you configure the `load-scripts-from-flash` statement at the `[edit system scripts]` hierarchy level, Junos OS creates the **/config/scripts/lib/** directory. When you add or remove the `load-scripts-from-flash` statement in the configuration, you must manually move scripts and script libraries from the hard disk to the flash drive, or vice versa, as appropriate. They are not moved automatically.

Imported scripts must be stored in the **/var/db/scripts/lib/** directory on the hard disk, or if the `load-scripts-from-flash` statement is configured, in the **/config/db/scripts/lib/** directory on the flash drive. To import a script from the script library, include the `<xsl:import>` tag in the style sheet declaration of an *XSLT* script or the `import` statement in a SLAX script and specify the file location. The following sample code imports the **/var/db/scripts/lib/test.xsl** file:

XSLT Syntax

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

SLAX Syntax

```
version 1.2;
import "../lib/test.xsl";
```

RELATED DOCUMENTATION

Store and Enable Junos Automation Scripts | 448
Store Scripts in Flash Memory | 451

CHAPTER 16

Configure a Remote Source for Scripts

IN THIS CHAPTER

Overview of Updating Scripts from a Remote Source

You can update the automation scripts on Junos devices by retrieving a copy from a remote machine (which can be another Junos device or a regular networked computer). This eases file management, because you can make changes to the master script in a single location and then update the copy on each device where the script is currently enabled. Each device continues to use its locally stored scripts, only updating a script when you issue the appropriate operational or configuration mode command.

For each script, you can configure the `source` statement and a URL at the hierarchy level where you configured the script to define the remote location that houses the master copy of that script. When you then issue the `set refresh` configuration mode command for a script, the device updates its local copy by retrieving the 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. To refresh a single script or multiple scripts from the remote location, you issue the `set refresh-from` configuration mode command at the appropriate hierarchy level and specify the URL. You can also refresh a single script from a remote location using the `request system scripts refresh-from` operational mode command.

You can use the `set refresh` and `set refresh-from` commands to update either an individual script or all scripts of a given type on the device. When you issue the `set refresh` or `set refresh-from` command, the device immediately attempts to connect to the appropriate remote source for each script. If successful, the device updates the local script with the remote source. If the device encounters a problem, it generates an error message.

Issuing the `set refresh` or `set refresh-from` command does not add the `refresh` and `refresh-from` statements to the configuration. Thus, these commands behave like operational mode commands by executing an operation, instead of adding a statement to the configuration. The `refresh` and `refresh-from` statements are mutually exclusive.

If a device has dual Routing Engines and you want to update the script on both Routing Engines, you must issue the `set refresh` or `set refresh-from` command on each Routing Engine separately. Alternatively, you can refresh the scripts on the requesting Routing Engine and then use either the `request system scripts synchronize` operational mode command to synchronize scripts to the other Routing Engine or the `commit synchronize scripts` configuration mode command to synchronize all scripts to the other Routing Engine when you commit and synchronize the configuration. If you use the `request system scripts refresh-from` operational mode command to refresh a script from a specific URL, include the `sync` option to refresh the script on both Routing Engines.

CAUTION: For commit scripts, 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. 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 "Use an Alternate Source Location for a Script" on page 467.

RELATED DOCUMENTATION

Configure and Use a Master Source Location for a Script | 457

Configure and Use a Master Source Location for a Script

IN THIS SECTION

You can store a master copy of each script in a central repository. This eases file management because you can make changes to the master script in one place and then update the copy on each device where the script is currently enabled. This section discusses how to configure the master source location for a script and how to refresh the local copy of the script.

Configuring the Master Source for a Script

To specify the source location for the master copy of an individual script:

1. Configure the `source` statement, and specify the URL of the master file as an HTTPS URL, HTTP URL, FTP URL, or secure copy (scp)-style remote file specification.

   The configuration hierarchy depends on the script type and filename.

   ```
   [edit event-options event-script file filename]
   user@host# set source url
   ```

   ```
   [edit system extensions extension-service application file filename]
   user@host# set source url
   ```

   ```
   [edit system scripts (commit | op | snmp) file filename]
   user@host# set source url
   ```

The following example configures the master source location for the **ospf-neighbors.slax** op script.

```
[edit system scripts op file ospf-neighbors.slax]
user@host# set source https://example.juniper.net/scripts/ospf-neighbors.slax
```

2. If the source location is on an HTTPS server, configure the cert-file statement on supported devices. Specify the absolute path to the certificate (Root CA or self-signed) in Privacy-Enhanced Mail (PEM) format that the device uses to validate the server's certificate.

   The configuration hierarchy depends on the script type and filename.

```
[edit event-options event-script file filename]
user@host# set cert-file path
```

```
[edit system extensions extension-service application file filename]
user@host# set cert-file path
```

```
[edit system scripts (commit | op | snmp) file filename]
user@host# set cert-file path
```

   The following example configures the Root CA certificate that will be used to validate the certificate of the server where the master copy of the **ospf-neighbors.slax** op script resides.

```
[edit system scripts op file ospf-neighbors.slax]
user@host# set cert-file /var/tmp/root-ca-cert-example-juniper-net.pem
```

Including the source statement in the configuration does not affect the local copy of the script until you issue the set refresh command. At that point, the device retrieves the master copy from the specified URL and overwrites the local copy.

Configuring the Routing Instance Used to Update a Script from the Master Source

Before Junos OS Release 18.1R1, scripts could be updated from a master source using the default management interface. However, starting in Junos OS Release 17.3R1, you are able to confine the management interface in a nondefault virtual routing and forwarding (VRF) instance so that management traffic no longer has to share a routing table with other control or protocol traffic. Starting in Junos OS Release 18.1R1, you can specify a routing instance to use to refresh or download a commit, event, JET, op, or SNMP script from a master source. This routing instance can be either the nondefault management instance mgmt_junos or some other routing instance.

You must configure the routing instance you specify to update scripts through at the `[edit system routing-instances]` hierarchy level.

To use `mgmt_junos` to update scripts from a master source:

1. Enable mgmt_junos by configuring the `management-instance` statement at the `[edit system]` hierarchy level.

```
user@host# set system management-instance
```

2. Configure the mgmt_junos routing instance at the `[edit routing-instances]` hierarchy level.

```
user@host# set routing-instances mgmt_junos description description
```

3. Configure the mgmt_junos routing instance at one of the three `routing-instance` hierarchy levels for scripts:

   - For commit, op, or SNMP scripts, configure the mgmt_junos routing instance at the `[edit system scripts (commit | op | snmp)]` hierarchy level.

   ```
   user@host# set system scripts (commit | op | snmp) file filename routing-instance
   mgmt_junos
   ```

   - For event scripts, configure the mgmt_junos routing instance at the `[edit event-options event-script file]` hierarchy level.

   ```
   user@host# set event-options event-script file filename routing-instance mgmt_junos
   ```

   - For JET scripts, configure the mgmt_junos routing instance at the `[edit system extensions extension-service application file]` hierarchy level.

   ```
   user@host# set system extensions extension-service application file filename routing-
   instance mgmt_junos
   ```

NOTE: To update scripts from a master source using a configured management interface, you can configure only `mgmt_junos` for the *`routing-instance-name`*. To use a non-management interface, you can configure anything for the *`routing-instance-name`*.

SEE ALSO

routing-instance (JET Scripts)

routing-instance (System Scripts)

routing-instance (Event Scripts)

Updating a Script from the Master Source

If you configure a master source for one or more scripts on a device, you can refresh the scripts on that device using the `set refresh` configuration mode command. You can update a single script or all scripts of a given script type that have a master source location configured.

The update operation occurs as soon as you issue the `set refresh` command. When you issue the `set refresh` command, the device immediately attempts to connect to the specified URL and retrieve a copy of the master file. The master copy overwrites the local script stored in the scripts directory on the device. If the `load-scripts-from-flash` statement is configured, the device updates the script on the flash drive instead of the script on the hard disk. If a master source is not defined for a script, that script is not updated and a warning is issued. For commit scripts, the updated *commit script* is executed when you next issue the `commit` command. If the script configuration includes the `routing-instance` statement, then Junos OS updates the script using that routing instance.

> **NOTE**: Issuing the `set refresh` command does not add the `refresh` statement to the configuration. Thus the command behaves like an operational mode command by executing an operation, instead of adding a statement to the configuration.
>
> The `set refresh` command is unique in the Junos OS CLI in that it behaves like an operational mode command and yet it can be executed from within configuration mode. All other Junos OS CLI operational mode commands can only be executed from command mode. The functionality is provided in this manner as a convenience to users developing commit scripts.

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. Alternatively, you can refresh the scripts on the requesting Routing Engine and then use either the `request system scripts synchronize` operational mode command to synchronize scripts to the other Routing Engine or the `commit synchronize scripts` configuration mode command to synchronize all scripts to the other Routing Engine when you commit and synchronize the configuration.

To update a single script from its master source, issue the `set refresh` command at the hierarchy level where the script is configured. The hierarchy location depends on the script type and filename as shown in the following examples. The `source` statement specifying the master source location must already be configured.

```
[edit event-options event-script file filename]
user@R1# set refresh
```

```
[edit system extensions extension-service application file filename]
user@R1# set refresh
```

```
[edit system scripts commit file filename]
user@R1# set refresh
```

```
[edit system scripts op file filename]
user@R1# set refresh
```

```
[edit system scripts snmp file filename]
user@R1# set refresh
```

To update all enabled scripts of a given script type from their master source files, issue the `set refresh` command at the hierarchy level for that script type.

```
[edit event-options event-script]
user@R1# set refresh
```

```
[edit system extensions extension-service application]
user@R1# set refresh
```

```
[edit system scripts commit]
user@R1# set refresh
```

```
[edit system scripts op]
user@R1# set refresh
```

```
[edit system scripts snmp]
user@R1# set refresh
```

SEE ALSO

Example: Configure and Refresh from the Master Source for a Script | 463
Use an Alternate Source Location for a Script | 467
refresh (Commit Scripts)
refresh (Op Scripts)
refresh (Event Scripts)
refresh (SNMP Scripts)

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
21.2R1	Starting in Junos OS Release 21.2R1, you must specify a certificate when you refresh scripts from an HTTPS server.

Example: Configure and Refresh from the Master Source for a Script

IN THIS SECTION

The following example configures a master source file for an op script on a device running Junos OS. The remote source is defined as an HTTP URL. The example uses the master source to update the local copy of the script on the device.

Requirements

- Routing, switching, or security device running Junos OS.

Overview

You can store a master copy of each script in a central repository. You can make changes to the master script in one place and then update the local copy of the script on devices where the script is enabled.

This example enables the op script **iso.xsl** on a device running Junos OS and then configures a master source location for the script. The remote source for the **iso.xsl** file is the HTTP URL http://my.example.com/pub/scripts/iso.xsl.

Once you configure the master source location, you refresh the local script by issuing the `set refresh` configuration mode command at the hierarchy level where you configured the script. In this example, you would issue the `set refresh` command at the `[edit system scripts op file iso.xsl]` hierarchy level.

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and configure the master source location for the script:

1. Copy the script to the **/var/db/scripts/op/** directory on the device.

2. In configuration mode, configure the `file` statement to enable the **iso.xsl** script.

```
[edit system scripts op]
user@R1# set file iso.xsl
```

3. To configure the master source for the **iso.xsl** file, include the `source` statement and source location at the `[edit system scripts op file iso.xsl]` hierarchy level.

```
[edit system scripts op file iso.xsl]
user@R1# set source http://my.example.com/pub/scripts/iso.xsl
```

4. Issue the `commit and-quit` command to commit the configuration and exit to operational mode.

```
[edit]
user@R1# commit and-quit
```

Results

```
system {
    scripts {
        op {
            file iso.xsl {
                source http://my.example.com/pub/scripts/iso.xsl;
            }
        }
    }
}
```

Verifying the Script

Purpose

Verify that the script is on the device and enabled in the configuration.

Action

Issue the `file list` operational mode command to view the files in the specified directory. The `detail` option provides additional information such as permissions, file size, and modified date.

```
user@R1> file list /var/db/scripts/op detail

/var/db/scripts/op:
total 128
-rw-r--r--  1 root  admin  13897 Feb 10  2011 iso.xsl
...
```

Issue the `show configuration system scripts op` operational mode command to list the op scripts currently enabled on the device.

```
user@R1> show configuration system scripts op
file iso.xsl
```

Refreshing the Script from the Master Source

Step-by-Step Procedure

To refresh the local copy of the script from the master source file:

1. In configuration mode, issue the `set refresh` command at the `[edit system scripts op file iso.xsl]` hierarchy level.

```
[edit system scripts op file iso.xsl]
user@R1# set refresh
```

Verification

IN THIS SECTION

Verifying the Updated Script

Purpose

After refreshing the script, verify that the local copy is updated.

Action

Issue the `file list` operational mode command with the `detail` option to view the files in the specified directory. Verify that the modified date reflects the refreshed version.

```
user@R1> file list /var/db/scripts/op detail

/var/db/scripts/op:
total 128
-rw-r--r--  1 root  admin  14128 May 26  2011 iso.xsl
...
```

RELATED DOCUMENTATION

Use an Alternate Source Location for a Script

IN THIS SECTION

Refresh a Script from an Alternate Location

In addition to updating a script from the master source defined by the `source` statement, you also can update a script from an alternate location using the `set refresh-from` configuration mode command or the `request system scripts refresh-from` operational mode command. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems.

The update operation occurs as soon as you issue either the `set refresh-from` configuration mode command or the `request system scripts refresh-from` operational mode command. When you issue the command, the device immediately attempts to connect to the specified URL and retrieve a copy of the file. The copy overwrites the local script stored in the scripts directory on the device. If a copy of the source is not available at the remote URL, that script is not updated and a warning is issued. For commit scripts, the updated commit script is executed when you next issue the `commit` command.

Issuing the `set refresh-from` command does not add the `refresh-from` statement to the configuration. Thus the `set refresh-from` command 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 to update the script on both Routing Engines, you must issue the `set refresh-from` command on each Routing Engine separately. Alternatively, you can refresh the scripts on the requesting Routing Engine and then use either the `request system scripts` `synchronize` operational mode command to synchronize scripts to the other Routing Engine or the `commit synchronize scripts` configuration mode command to synchronize all scripts to the other Routing Engine when you commit and synchronize the configuration. In operational mode, you can also use the `request system scripts refresh-from` command with the `sync` option to refresh the script on both Routing Engines.

To update one or more scripts using the `set refresh-from` configuration mode command:

1. If any scripts will be refreshed from an HTTPS server, configure the `cert-file` statement for that script on supported devices. Specify the absolute path to the certificate (Root CA or self-signed) in Privacy-Enhanced Mail (PEM) format that the device will use to validate the server's certificate during the refresh operation.

   The hierarchy location for the `cert-file` statement depends on the script type and filename.

   ```
   [edit event-options event-script file filename]
   user@host# set cert-file path
   ```

   ```
   [edit system extensions extension-service application file filename]
   user@host# set cert-file path
   ```

   ```
   [edit system scripts (commit | op | snmp) file filename]
   user@host# set cert-file path
   ```

   The following example configures the Root CA certificate that will be used to validate the certificate of the server from which the **ospf-neighbors.slax** script is refreshed.

   ```
   [edit system scripts op file ospf-neighbors.slax]
   user@host# set cert-file /var/tmp/root-ca-cert-example-juniper-net.pem
   ```

2. If you configured certificates for any scripts in the previous step, commit the configuration.

   ```
   [edit]
   user@host# commit
   ```

3. To update a single script from an alternate source, issue the `set refresh-from` command under the hierarchy level where the script is configured, and specify the location of the remote file.

The hierarchy location depends on the script type and filename as shown in the following examples:

```
[edit event-options event-script file filename]
user@host# set refresh-from url
```

```
[edit system extensions extension-service application file filename]
user@R1# set refresh-from url
```

```
[edit system scripts commit file filename]
user@host# set refresh-from url
```

```
[edit system scripts op file filename]
user@host# set refresh-from url
```

```
[edit system scripts snmp file filename]
user@host# set refresh-from url
```

4. To instead update all enabled scripts of a given script type from an alternate source, issue the `set refresh-from` command at the hierarchy level for that script type, and specify the URL of the remote repository that houses the scripts.

```
[edit event-options event-script]
user@host# set refresh-from url
```

```
[edit system extensions extension-service application]
user@R1# set refresh-from url
```

```
[edit system scripts commit]
user@host# set refresh-from url
```

```
[edit system scripts op]
user@host# set refresh-from url
```

```
[edit system scripts snmp]
user@host# set refresh-from url
```

Alternatively, in operational mode, to update a single script from an alternate source, issue the `request system scripts refresh-from` command, and specify the script type, filename, and remote URL.

```
user@host> request system scripts refresh-from (commit | event | extension-service | op | snmp)
file filename url url <sync>
```

Where

- *filename*—Name of the script.
- *url*—URL of the remote script or directory. Specify the source as an HTTPS URL, HTTP URL, FTP URL, or secure copy (scp)-style remote file specification.

If you specify an HTTPS URL, you must also include the `cert-file` option in supported releases. The `cert-file` option specifies the path to the certificate (Root CA or self-signed) in PEM format that the device uses to validate the server's certificate. For example:

```
user@host> request system scripts refresh-from op file ospf-neighbors.slax url https://
example.juniper.net/scripts/ospf-neighbors.slax cert-file /var/tmp/root-ca-cert-example-juniper-
net.pem
refreshing 'ospf-neighbors.slax' from 'https://example.juniper.net/scripts/ospf-neighbors.slax'
/var/home/user/...transferring.file........100% of
```

If you request to refresh a script that does not exist at the remote site, the device generates an error message. For example:

```
user@host> request system scripts refresh-from op file nonexistent.slax url http://
host1.juniper.net/nonexistent.slax
refreshing 'nonexistent.slax' from 'http://host1.juniper.net/nonexistent.slax'
fetch-secure: http://host1.juniper.net/nonexistent.slax: Not Found
error: file-fetch failed
error: communication error: rpc failed (file-fetch)
error: error retrieving file http://host1.juniper.net/nonexistent.slax
```

When you issue the `set refresh-from` command, Junos OS creates a temporary folder, which is used for the file transfer. After the transfer and refresh operations are complete, Junos OS deletes the temporary folder.

SEE ALSO

request system scripts refresh-from

refresh-from (Commit Scripts)

refresh-from (Op Scripts)

refresh-from (Event Scripts)

Configuring the Routing Instance Used to Update a Script from an Alternate Source

Before Junos OS Release 18.1R1, scripts could be updated from a alternate source using the default management interface. However, starting in Junos OS Release 17.3R1, you are able to confine the management interface in a nondefault virtual routing and forwarding (VRF) instance so that management traffic no longer has to share a routing table with other control or protocol traffic. Starting in Junos OS Release 18.1R1, you can specify a routing instance to use to refresh or download a commit, event, JET, op, or SNMP script from a alternate source. This routing instance can be either the nondefault management instance `mgmt_junos` or some other routing instance.

You must configure the routing instance you specify to update scripts through at the `[edit system routing-instances]` hierarchy level.

To use a non-management routing instance to update scripts from an alternate source:

1. Configure the routing instance at the `[edit routing-instances]` hierarchy level.

   ```
   user@host# set routing-instances routing-instance-name description description
   ```

2. Configure the same routing instance at one of the three `routing-instance` hierarchy levels for scripts:

   - For commit, op, or SNMP scripts, configure the routing instance at the `[edit system scripts (commit | op | snmp)]` hierarchy level.

     ```
     user@host# set system scripts (commit | op | snmp) file filename routing-instance routing-instance-name
     ```

   - For event scripts, configure the routing instance at the `[edit event-options event-script file]` hierarchy level.

     ```
     user@host# set event-options event-script file filename routing-instance routing-instance-name
     ```

   - For JET scripts, configure the routing instance at the `[edit system extensions extension-service application file]` hierarchy level.

     ```
     user@host# set system extensions extension-service application file filename routing-instance routing-instance-name
     ```

NOTE: To update scripts from an alternate source using a configured management interface, you can configure only `mgmt_junos` for the *routing-instance-name*. To use a non-management interface, you can configure anything for the *routing-instance-name*.

SEE ALSO

routing-instance (JET Scripts)

routing-instance (System Scripts)

routing-instance (Event Scripts)

Example: Refresh a Script from an Alternate Source

IN THIS SECTION

The following example uses an alternate source location to update the local copy of the script on a device running Junos OS. The remote source is defined as an HTTP URL.

Requirements

- Routing, switching, or security device running Junos OS.

Overview

You can update a script from a location other than that of the master source. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems. You can refresh a single script or all scripts of a given type from the alternate location.

This example enables the op script **iso.xsl** on a device running Junos OS and then refreshes the script from a location other than the master source location. The remote source for the **iso.xsl** file is the HTTP URL http://my.example.com/pub/scripts2/iso.xsl.

You refresh the local script by issuing the `set refresh-from` configuration mode command at the hierarchy level where you configured the script. In this example, you would issue the `set refresh-from` command at the `[edit system scripts op file iso.xsl]` hierarchy level.

Configuration

IN THIS SECTION

Download and Enable the Script

Step-by-Step Procedure

To download and enable the script:

1. Copy the script to the **/var/db/scripts/op/** directory on the device.

2. In configuration mode, configure the `file` statement to enable the **iso.xsl** script.

```
[edit system scripts op]
user@R1# set file iso.xsl
```

3. Issue the `commit and-quit` command to commit the configuration and exit to operational mode.

```
[edit]
user@R1# commit and-quit
```

Results

```
system {
    scripts {
        op {
            file iso.xsl;
        }
```

```
    }
}
```

Verify the Script

Purpose

Verify that the script is on the device and enabled in the configuration.

Action

Issue the `file list` operational mode command to view the files in the specified directory. The `detail` option provides additional information such as permissions, file size, and modified date.

```
user@R1> file list /var/db/scripts/op detail

/var/db/scripts/op:
total 128
-rw-r--r--  1 root  admin  13897 Feb 10  2011 iso.xsl
...
```

Issue the `show configuration system scripts op` operational mode command to list the op scripts currently enabled on the device.

```
user@R1> show configuration system scripts op
file iso.xsl
```

Refresh the Script from the Alternate Location

Step-by-Step Procedure

To refresh the local copy of the script from the alternate location:

- In configuration mode, issue the `set refresh-from` command at the `[edit system scripts op file iso.xsl]` hierarchy level.

  ```
  [edit system scripts op file iso.xsl]
  user@R1# set refresh-from http://my.example.com/pub/scripts2/iso.xsl
  ```

Verify the Updated Script

Purpose

After refreshing the script, verify that the local copy is updated.

Action

Issue the `file list` operational mode command with the `detail` option to view the files in the specified directory. Verify that the modified date reflects the refreshed version.

```
user@R1> file list /var/db/scripts/op detail

/var/db/scripts/op:
total 128
-rw-r--r--  1 root  admin  14128 May 26  2011 iso.xsl
...
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
21.2R1	Starting in Junos OS Release 21.2R1, you must specify a certificate when you refresh scripts from an HTTPS server.

RELATED DOCUMENTATION

Configure and Use a Master Source Location for a Script | 457

request system scripts refresh-from

refresh-from (Commit Scripts)

refresh-from (Op Scripts)

refresh-from (Event Scripts)

CHAPTER 17

Configure the Session Protocol for Scripts

IN THIS CHAPTER

Understanding the Session Protocol in Automation Scripts

The Junos XML management protocol is a Juniper Networks proprietary protocol that is used to request information from and configure devices running Junos OS. The NETCONF XML management protocol is a standard that is used to request and change configuration information on a routing, switching, or security device. The NETCONF protocol is defined in RFC 6241, *NETCONF Configuration Protocol*, which is available at http://www.ietf.org/rfc/rfc6241.txt .

The `jcs:open()` function, which is used in SLAX and XSLT scripts to establish a connection with a device, includes the option to create a session either with the Junos XML protocol server on devices running Junos OS or with the NETCONF server on devices where NETCONF service over *SSH* is enabled. Prior to Junos OS Release 11.4, the function supported only sessions with the Junos XML protocol server on devices running Junos OS. The additional support for NETCONF sessions enables automation scripts to configure and manage devices in a multi-vendor environment.

The `jcs:open()` function supports the following session protocol types:

- `junoscript`—Session with the Junos XML protocol server on a routing, switching, or security device running Junos OS. This session type supports the operations defined in the Junos XML protocol and the Junos XML API, which are used to configure devices running Junos OS or to request information about the device configuration or operation. This is the default session type.

- `netconf`—Session with the NETCONF XML protocol server on a routing, switching, or security device over an SSHv2 connection. The device to which the connection is made must be enabled for NETCONF service over SSH. NETCONF over SSH is described in RFC 4742, *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, which is available at http://www.ietf.org/rfc/rfc4742.txt .

- `junos-netconf`—Proprietary session with the NETCONF XML protocol server over an SSHv2 connection on a routing, switching, or security device running Junos OS.

The NETCONF server on a device running Junos OS has the additional capabilities defined in `http://xml.juniper.net/netconf/junos/1.0` . The NETCONF server on these devices supports NETCONF XML protocol operations, most Junos XML protocol operations, and the tag elements defined in the Junos XML API. For `netconf` and `junos-netconf` sessions with devices running Junos OS, you should use only native NETCONF XML protocol operations and the extensions available in the Junos XML protocol for configuration functions as documented in the NETCONF XML Management Protocol Developer Guide.

The syntax for the `jcs:open()` function when specifying a session protocol is:

SLAX Syntax

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

XSLT Syntax

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

The *session-options* parameter is an XML node-set that specifies the session type and connection parameters. The session type is one of three values: `junoscript`, `netconf`, or `junos-netconf`. If you do not specify a session type, the default is `junoscript`, which opens a session with the Junos XML protocol server on a device running Junos OS. The format of the node-set is:

```
var $session-options := {
           <method> ("junoscript" | "netconf" | "junos-netconf");
           <username> "username";
           <passphrase> "passphrase";
           <password> "password";
           <port> "port-number";
           <instance> "routing-instance-name";
           <routing-instance> "routing-instance-name";
       }
```

If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script. The `<passphrase>` and `<password>` elements serve the same purpose. If you do not specify a passphrase or password element and it is required for authentication, you should be prompted for one during script execution by the device to which you are connecting.

NOTE: Devices running Junos OS Evolved support only password-less login when establishing a local or remote connection using the `jcs:open()` function. They do not support supplying a password as a function argument or using an interactive password as is supported by Junos OS. To effect a local or remote connection, execute this functionality by way of a password-less login or else authentication issues could be encountered.

Optionally, you can specify the server port number for `netconf` and `junos-netconf` sessions. The default NETCONF server port number is 830. If you do not specify a port number for a `netconf` or `junos-netconf` session, `jcs:open()` connects to the NETCONF server using port 830. However, if you specify a port number, `jcs:open()` connects to the given port instead. Specifying a port number has no impact on `junoscript` sessions, which are always established over SSH port 22.

To redirect the SSH connection to originate from within a specific routing instance, include the `instance` or `routing-instance` element and the routing instance name. The routing instance must be configured at the `[edit routing-instances]` hierarchy level. The remote device must be reachable either using the routing table for that routing instance or from one of the interfaces configured under that routing instance. The `instance` and `routing-instance` elements serve the same purpose.

To verify the protocol for a specific connection, call the `jcs:get-protocol(`*`connection`*`)` extension function and pass the connection handle as the argument. The function returns “junoscript”, “netconf”, or “junos-netconf”, depending on the session type.

During session establishment with a NETCONF server, the client application and NETCONF server each emit a `<hello>` tag element to specify which operations, or *capabilities,* they support from among those defined in the NETCONF specification or published as proprietary extensions. In `netconf` and `junos-netconf` sessions, you can retrieve the session capabilities of the NETCONF server by calling the `jcs:get-hello(`*`connection`*`)` extension function.

For example, the NETCONF server on a typical device running Junos OS might return the following capabilities:

```
<hello>
  <capabilities>
     <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
     </capability>
     <capability>
```

```
        urn:ietf:params:xml:ns:netconf:capability:validate:1.0
      </capability>
      <capability>
        urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
      </capability>
      <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
      <capability>http://xml.juniper.net/dmi/system/1.0</capability>
   </capabilities>
   <session-id>20826</session-id>
</hello>
```

RELATED DOCUMENTATION

Example: Specify the Session Protocol for a Connection within Scripts

IN THIS SECTION

The following example demonstrates how to specify the session protocol within a Junos automation script when creating a connection with a remote device. Specifically, the example *op script* establishes a

NETCONF session with a remote device running Junos OS, retrieves and prints the NETCONF server capabilities, and then updates and commits the configuration on that device.

Requirements

- Routing, switching, or security device running Junos OS Release 11.4 or later.
- Client application can log in to the device where the NETCONF server resides.
- NETCONF service over *SSH* is enabled on the device where the NETCONF server resides.

Overview and Script

The `jcs:open()` extension function includes the option to create a session either with the Junos XML protocol server on devices running Junos OS or with the NETCONF server on devices where NETCONF service over SSH is enabled. In the following example, the script creates a connection and establishes a NETCONF session with a remote device running Junos OS. If the connection and session are successfully established, the script updates the configuration on the remote device to add the `ftp` statement to the `[edit system services]` hierarchy level. The script also retrieves and prints the session protocol and the capabilities of the NETCONF server.

The script takes one argument, `remote-host`, which is the IP address or hostname of the remote device. The `arguments` variable is declared at the global level of the script so that the argument name and description are visible in the command-line interface (CLI) when a user requires *context-sensitive help*.

The variable `netconf` is a node-set that specifies the session protocol and the connection parameters for the remote device. The value of the `<method>` element is set to “netconf” to establish a session with the NETCONF server over an SSHv2 connection. The `<username>` element specifies the username for the connection. If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script. In this example, the passphrase and port are not specified. If a passphrase is required for authentication, the remote device should prompt for one during script execution. The script establishes the session using the default NETCONF port 830.

If the connection and establishment of the NETCONF session are successful, the script executes remote procedure calls (RPCs). The RPCs contain the tag elements `<lock>`, `<edit-config>`, `<commit>`, and `<unlock>`, which are NETCONF operations to lock, edit, commit, and unlock the *candidate configuration*. The script stores the *RPC* for each task in a separate variable. The results for each RPC are also stored separately and parsed for errors. The script only executes each subsequent step if the previous step is successful. For example, if the script cannot lock the configuration, it does not execute the RPCs to edit, commit, or unlock the configuration.

The variable `rpc-edit-config` contains the tag element `<edit-config>`, which is a NETCONF operation to modify a configuration. The child element, `<config>`, includes the modified portion of the configuration that is merged with the candidate configuration on the device. If errors are encountered, the script calls the `copy-of` statement to copy the result tree fragment variable to the results tree so that the error message prints to the CLI during script execution.

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";
ns ext = "http://xmlsoft.org/XSLT/namespace";

var $arguments = {
     <argument> {
          <name> "remote-host";
            <description> "device hostname or IP address to which to connect";
     }
}
param $remote-host;

match / {

    <op-script-results> {

       var $netconf := {
           <method> "netconf";
           <username> "bsmith";
       }

       var $rpc-lock-config = {
          <lock> {
             <target> {
                <candidate>;
             }
          }
       }

       var $rpc-unlock-config = {
          <unlock> {
             <target> {
                <candidate>;
             }
          }
       }
```

```
    var $rpc-commit = {
        <commit>;
    }

    var $rpc-edit-config = {
        <edit-config> {
            <target> {
                <candidate>;
            }
            <default-operation> "merge";
            <config> {
                <configuration> {
                    <system> {
                        <services> {
                            <ftp>;
                        }
                    }
                }
            }
        }
    }

    if ($remote-host = '') {
        <xnm:error> {
            <message> "missing mandatory argument 'remote-host'";
        }
    }
    else {

        var $connection = jcs:open($remote-host, $netconf);
        if ($connection) {

            /* request protocol and capabilities */
            var $protocol = jcs:get-protocol($connection);
            var $capabilities = jcs:get-hello($connection);

            <output> "\nSession protocol: " _ $protocol _ "\n";
            copy-of $capabilities;

            /* execute rpcs to lock, edit, commit, and unlock config */
            var $lock-reply = jcs:execute($connection, $rpc-lock-config);
            if ($lock-reply/..//rpc-error) {
                copy-of $lock-reply;
```

```
                }
                else {
                   var $edit-config-reply = jcs:execute($connection, $rpc-edit-config);
                   if ($edit-config-reply/..//rpc-error) {
                      <output>"Configuration error: "_ $edit-config-reply/..//error-message/.
                                    _ "\nConfiguration not committed.\n" ;
                      copy-of $edit-config-reply;
                   }
                   else {
                      var $commit-reply = jcs:execute($connection, $rpc-commit);
                      if ($commit-reply/..//rpc-error) {
                         <output>"Commit error or warning: " _ $commit-reply/..//error-message/.;
                         copy-of $commit-reply;
                      }
                   }
                   var $unlock-reply = jcs:execute($connection, $rpc-unlock-config);
                }

                expr jcs:close($connection);
             }
             else {
                <output>"\nNo connection - exiting script";
             }

          }
       }
    }
```

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"
    xmlns:ext="http://xmlsoft.org/XSLT/namespace" version="1.0">

  <xsl:variable name="arguments">
    <argument>
      <name>remote-host</name>
      <description>device hostname or IP address to which to connect</description>
    </argument>
```

```
</xsl:variable>

<xsl:param name="remote-host"/>

<xsl:template match="/">
  <op-script-results>
    <xsl:variable name="netconf-temp-1">
      <method>netconf</method>
      <username>bsmith</username>
    </xsl:variable>
    <xsl:variable xmlns:ext="http://xmlsoft.org/XSLT/namespace"
      name="netconf" select="ext:node-set($netconf-temp-1)"/>

    <xsl:variable name="rpc-lock-config">
      <lock>
        <target>
          <candidate/>
        </target>
      </lock>
    </xsl:variable>

    <xsl:variable name="rpc-unlock-config">
      <unlock>
        <target>
          <candidate/>
        </target>
      </unlock>
    </xsl:variable>

    <xsl:variable name="rpc-commit">
      <commit/>
    </xsl:variable>

    <xsl:variable name="rpc-edit-config">
      <edit-config>
        <target>
          <candidate/>
        </target>
        <default-operation>merge</default-operation>
        <config>
          <configuration>
            <system>
              <services>
```

```
            <ftp/>
          </services>
        </system>
      </configuration>
    </config>
  </edit-config>
</xsl:variable>

<xsl:choose>
  <xsl:when test="$remote-host = ''">
    <xnm:error>
      <message>missing mandatory argument 'remote-host'</message>
    </xnm:error>
  </xsl:when>
  <xsl:otherwise>
    <xsl:variable name="connection" select="jcs:open($remote-host, $netconf)"/>

    <xsl:choose>
      <xsl:when test="$connection">

        <!-- request protocol and capabilities -->
        <xsl:variable name="protocol" select="jcs:get-protocol($connection)"/>
        <xsl:variable name="capabilities" select="jcs:get-hello($connection)"/>
        <output>
          <xsl:value-of select="concat('&#10;Session protocol: ', $protocol, '&#10;')"/>
        </output>
        <xsl:copy-of select="$capabilities"/>

        <!-- execute rpcs -->
        <xsl:variable name="lock-reply"
          select="jcs:execute($connection, $rpc-lock-config)"/>
        <xsl:choose>
          <xsl:when test="$lock-reply/..//rpc-error">
            <xsl:copy-of select="$lock-reply"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:variable name="edit-config-reply"
             select="jcs:execute($connection, $rpc-edit-config)"/>
            <xsl:choose>
              <xsl:when test="$edit-config-reply/..//rpc-error">
                <output>
                  <xsl:value-of select="concat('Configuration error: ',
                    $edit-config-reply/..//error-message/.,
```

```
                        '&#10;Configuration not committed.&#10;')"/>
                    </output>
                    <xsl:copy-of select="$edit-config-reply"/>
                  </xsl:when>
                  <xsl:otherwise>
                    <xsl:variable name="commit-reply"
                      select="jcs:execute($connection, $rpc-commit)"/>
                    <xsl:if test="$commit-reply/..//rpc-error">
                      <output>
                        <xsl:value-of select="concat('Commit error or warning: ',
                         $commit-reply/..//error-message/.)"/>
                      </output>
                      <xsl:copy-of select="$commit-reply"/>
                    </xsl:if>
                  </xsl:otherwise>
                </xsl:choose>
                <xsl:variable name="unlock-reply" select="jcs:execute($connection,
                  $rpc-unlock-config)"/>
              </xsl:otherwise>
            </xsl:choose>

            <xsl:value-of select="jcs:close($connection)"/>
          </xsl:when>
          <xsl:otherwise>
            <output>No connection - exiting script</output>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:otherwise>
    </xsl:choose>
  </op-script-results>
 </xsl:template>
</xsl:stylesheet>
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the XSLT or SLAX script into a text file, name the file **netconf-session.xsl** or **netconf-session.slax** as appropriate, and copy it to the **/var/db/scripts/op/** directory on the device.

2. In configuration mode, include the `file` statement at the `[edit system scripts op]` hierarchy level and **netconf-session.xsl** or **netconf-session.slax** as appropriate.

```
[edit system scripts op]
bsmith@local-host# set file netconf-session.(slax | xsl)
```

3. Issue the `commit and-quit` command.

```
[edit]
bsmith@local-host# commit and-quit
```

4. Execute the op script on the local device by issuing the `op netconf-session` operational mode command and include any necessary arguments.

   In this example, the user, bsmith, is connecting to the remote device, fivestar. The remote device has dual routing engines, so the `commit` operation returns a warning that the `commit synchronize` command should be used to commit the new candidate configuration to both routing engines.

```
bsmith@local-host> op netconf-session remote-host fivestar
bsmith@fivestar's password:
Session protocol: netconf
Commit error or warning:
graceful-switchover is enabled, commit synchronize should be used
```

Verification

IN THIS SECTION

Confirm that the device is working properly.

Verifying Op Script Execution

Purpose

Verify that the script behaves as expected.

Action

Review the script output in the CLI and in the op script log file. Take particular note of any errors that occurred during execution. The default op script log file is **/var/log/op-script.log**. If the log file is significantly lengthy, limit the display by appending the `| last` *`number-of-lines`* option to the `show log` command and specify the number of lines to print to the CLI. The output within the `<op-script-results>` element is relevant to the script execution.

```
bsmith@local-host> show log op-script.log | last 100
...output omitted for brevity...
<op-script-results 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">
<output>
Session protocol: netconf
</output>
<hello>
  <capabilities>
   <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
   <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
   </capability>
   <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
   </capability>
   <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>
   <capability>
      urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
   </capability>
```

```
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>29087</session-id>
</hello>
  <output>Commit error or warning:
graceful-switchover is enabled, commit synchronize should be used
</output>
  <rpc-error>
<error-severity>warning</error-severity>
<error-message>
graceful-switchover is enabled, commit synchronize should be used
</error-message>
</rpc-error>
  <ok/>
</op-script-results>
```

You can also obtain more descriptive script output on a device running Junos OS by including the `| display xml` option when you execute an op script.

```
bsmith@local-host> op netconf-session remote-host fivestar | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/11.4D0/junos">
  <output>
      Session protocol: netconf
  </output>
  <hello>
   <capabilities>
     <capability>
       urn:ietf:params:xml:ns:netconf:base:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:validate:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
     </capability>
```

```
      <capability>
        http://xml.juniper.net/netconf/junos/1.0
      </capability>
      <capability>
        http://xml.juniper.net/dmi/system/1.0
      </capability>
    </capabilities>
    <session-id>
        29087
    </session-id>
  </hello>
  <output>
      Commit error or warning:
      graceful-switchover is enabled, commit synchronize should be used
  </output>
  <rpc-error>
      <error-severity>
          warning
      </error-severity>
      <error-message>
          graceful-switchover is enabled, commit synchronize should be used
      </error-message>
  </rpc-error>
  <ok/>
</op-script-results>
<cli>
  <banner></banner>
</cli>
</rpc-reply>
```

Meaning

This example creates a NETCONF session on a remote device running Junos OS. The capabilities of the NETCONF server include both standard NETCONF operations and Juniper Networks proprietary extensions, which are defined in `http://xml.juniper.net/netconf/junos/1.0` and `http://xml.juniper.net/dmi/system/1.0`. The RPC results for the `commit` operation include one warning, but the commit operation is still successful.

Verifying the Configuration Changes

Purpose

Verify that the commit was successful by viewing the configuration change and the commit log on the remote device.

Action

On the remote device, execute the `show configuration system services` operational mode command to view the `[edit system services]` hierarchy level of the configuration. If the script is successful, the configuration includes the `ftp` statement.

```
bsmith@fivestar> show configuration system services
ftp;
netconf {
    ssh;
}
```

Additionally, you can review the commit log. On the remote device, execute the `show system commit` operational mode command to view the commit log. In this example, the log confirms that bsmith committed the candidate configuration in a NETCONF session at the given date and time.

```
bsmith@fivestar> show system commit
0   2011-07-11 12:04:01 PDT by bsmith via netconf
1   2011-07-08 15:16:33 PDT by root via cli
```

Troubleshooting

IN THIS SECTION

Troubleshooting Connection Errors

Problem

The script generates the following error message:

```
hello packet:1:(0) Document is empty
hello packet:1:(0) Start tag expected, '<' not found
error: netconf: could not read hello
error: did not receive hello packet from server
error: Error in creating the session with "fivestar" server
No connection - exiting script
```

Potential causes for the connection error include:

- The device or interface to which you are connecting is down or unavailable.
- The script argument for the IP address or DNS name of the remote device is incorrect.
- The connection timeout value was exceeded before establishing the connection.
- The user authentication for the remote device is not valid or is entered incorrectly.
- You are trying to establish a NETCONF session, and NETCONF over SSH is not enabled on the device where the NETCONF server resides, or it is enabled on a different port.

Solution

Ensure that the remote device is up and running and that the user has access to the device. Also verify that you supplied the correct argument for the IP address or DNS name of the remote device when executing the script.

For NETCONF sessions, ensure that you have enabled NETCONF over SSH on the device where the NETCONF server resides. Since the example program does not specify a specific port number for the NETCONF session, the session is established on the default NETCONF-over-SSH port, 830. To verify whether NETCONF over SSH is enabled on the default port for a device running Junos OS, enter the following operational mode command on the remote device:

```
bsmith@fivestar> show configuration system services

netconf {
```

```
    ssh;
}
```

If the netconf configuration hierarchy is absent on the remote device, issue the following statements in configuration mode to enable NETCONF over SSH on the default port:

```
[edit]
bsmith@fivestar# set system services netconf ssh
bsmith@fivestar# commit
```

If the netconf configuration hierarchy specifies a port other than the default port, include the port number in the XML node-set that you pass to the jcs:open() function. For example, the following device is configured for NETCONF over SSH on port 12345:

```
bsmith@fivestar> show configuration system services
netconf {
    ssh {
       port 12345;
    }
}
```

To create a NETCONF session on the alternate port, include the new port number in the XML node-set.

```
var $netconf := {
     <method> "netconf";
     <username> "bsmith";
     <port> "12345";
}
var $connection = jcs:open($remote-host, $netconf);
...
```

Troubleshooting Configuration Lock Errors

Problem

The script generates one of the following error messages:

```
configuration database locked by:
  root terminal p0 (pid 24113) on since 2011-07-11 11:48:06 PDT, idle 00:07:59
```

```
Users currently editing the configuration:
  root terminal p1 (pid 24279) on since 2011-07-11 12:28:30 PDT
      {master}[edit]
```

```
configuration database modified
```

Solution

Another user currently has a lock on the candidate configuration or has modified the candidate configuration but has not yet committed the configuration. Wait until the lock is released, and then execute the program.

Troubleshooting Configuration Syntax Errors

Problem

The following error message prints to the CLI:

```
Configuration error: syntax error
Configuration not committed.
```

Examine the result tree for additional information. In this case, the result tree shows the following error message:

```
  <rpc-error>
     <error-severity>
        error
     </error-severity>
```

```
    <error-info>
        <bad-element>
            ftp2
        </bad-element>
    </error-info>
    <error-message>
        syntax error
    </error-message>
</rpc-error>
```

Solution

The `<bad-element>` tag element indicates that the configuration statement is not valid. Correct the configuration hierarchy and run the script. In this example error, the user entered the tag `<ftp2>` instead of `<ftp>`. Since that is not an acceptable element in the configuration, the NETCONF server returns an error.

RELATED DOCUMENTATION

Understanding the Session Protocol in Automation Scripts | 477

https://www.juniper.net/documentation/en_US/junos/information-products/pathway-pages/junos-xml-management-protocol/junos-xml-management-protocol.html

https://www.juniper.net/documentation/en_US/junos/information-products/pathway-pages/netconf-guide/netconf.html

get-hello() Function (SLAX and XSLT) | 377

get-protocol() Function (SLAX and XSLT) | 381

open() Function (SLAX and XSLT) | 394

CHAPTER 18

Control Execution of Scripts

IN THIS CHAPTER

Configure Script Start Options

SUMMARY

Configure the device to perform specific system resource checks before executing a script.

IN THIS SECTION

You can configure start options for certain types of scripts on supported platforms. When you configure start options, the device performs specific system resource checks before executing a script. The checks determine whether the system should start or prevent script execution based on the configured values. By configuring script start options you can ensure that a device executes only essential scripts when system resources are limited, thereby enabling the device to continue performing all critical network functions.

You can configure start options to prevent script execution based on the system's memory usage, as described in "Configure the System Memory Usage Threshold for Scripts" on page 498.

You configure script start options to prevent a script from executing under certain conditions. To instead enable the device to start a script but slow the script's execution or halt the script if it exceeds certain limits as it runs, see the following documentation:

- "Dampen Script Execution" on page 509
- "Understanding Limits on Executed Event Policies and Memory Allocation for Scripts" on page 502

Configure the System Memory Usage Threshold for Scripts

You can configure a device to prevent a script from executing if the system's memory usage is above a configured threshold. For example, you might want to prevent the execution of a particularly resource intensive script or type of script if the system memory usage is above 75 percent. Alternatively, if a particular script plays a vital role in the device's operation, you might configure a higher system memory usage threshold or no threshold for that specific script to ensure that the device still executes the script even if the system has more limited resources at that time.

You configure the `start start-options mem-factor` statement to define the system memory usage threshold above which a script does not execute. You can define a threshold for op, event, SNMP, and Juniper Extension Toolkit (JET) scripts. You can apply the statement to an individual script or to all scripts of a given type. When you configure the statement for a type of script, for example, op scripts, the statement applies to all Python, SLAX, and XSLT scripts of that type. If you define different thresholds for the script type and an individual script of the same type, the configuration for the individual script takes precedence.

NOTE: Junos devices do not enforce script start options for remote op scripts, which are executed using the `op url` command. Script start options are enforced only for local op scripts.

NOTE: For JET applications, the `daemonize` and `mem-factor` statements are mutually exclusive. If a JET application includes the `daemonize` statement in its configuration, you cannot configure the `mem-factor` statement either for that individual script or globally for all JET scripts. If you configure both statements and commit the configuration, the device issues a commit error.

When you execute a script that has the `mem-factor` statement configured for that script or that type of script, the device compares the current system memory usage to the configured `mem-factor` value. If the system memory usage is lower than the configured threshold, then the script executes normally. If the system memory usage exceeds the configured threshold, the device does not execute the script and generates a system log message to that effect. If you do not configure the `mem-factor statement` for either the individual script or script type, then the system does not perform any system memory usage checks and executes the script normally.

Junos devices calculate the system memory usage differently depending on the operating system. Junos OS uses the following calculation to determine the system memory usage:

```
current memory usage = (Total memory - (Cache memory + Free memory + Inactive Memory))*100 /
Total memory
```

For example, consider the following `show system memory` command output on a device running Junos OS:

```
user@host> show system memory
System memory usage distribution:
       Total memory: 4135380 Kbytes (100%)
    Reserved memory:  137172 Kbytes (  3%)
       Wired memory:  334964 Kbytes (  8%)
      Active memory:  772468 Kbytes ( 18%)
    Inactive memory: 2738480 Kbytes ( 66%)
       Cache memory:       0 Kbytes (  0%)
        Free memory:  261964 Kbytes (  6%)
...
```

In this case, the current system memory usage is approximately twenty-seven percent. If you a invoke a script, the device does not execute the script if this usage exceeds the configured `mem-factor` value for that script. The device compares the usage against the `mem-factor` value for the script type only if you do not configure the `mem-factor` statement for the individual script.

Junos OS Evolved includes the system memory usage directly in the `show system memory` command's XML output.

```
user@host-re0 show system memory node re0 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/23.1R1/junos">
    <system-memory-information>
        <system-memory-summary-information>
            <system-memory-total>16062004</system-memory-total>
            <system-memory-total-percent>100%</system-memory-total-percent>
            <system-memory-used>3540932</system-memory-used>
            <system-memory-used-percent>22%</system-memory-used-percent>
...
```

When you invoke a script that has the `mem-factor` statement configured, the device logs the CSCRIPT_START_OPTIONS_MEM_FACTOR system log message with severity level INFO.

```
Oct 11 11:39:07  host cscript[93413]: CSCRIPT_START_OPTIONS_MEM_FACTOR: Start-options is set
with mem-factor '70'
```

If a device prevents the execution of a script because the current system memory usage exceeds the configured threshold, the cscript process records system log messages in the log file. The CLI does not issue any warning or error that the script was halted. The device logs the CSCRIPT_START_OPTIONS_SYSTEM_MEM and CSCRIPT_START_OPTIONS_WARNING system log messages with severity level WARNING for this event.

```
user@host> show log messages | match cscript
Oct 11 11:39:07  host cscript[93413]: CSCRIPT_START_OPTIONS_SYSTEM_MEM: Start-options System
Memory Usage Statistics: Total memory:'4137428' Free memory:'132936'
Oct 11 11:39:07  host cscript[93413]: CSCRIPT_START_OPTIONS_WARNING: Start-options System Memory
Usage '74' exceeds the mem-factor '70', aborting the script 'bgp-neighbors.slax' execution
```

NOTE: The CLI does not indicate if the device halts a script's execution. For example, op scripts do not emit any output in this case, SNMP scripts return a `No such instance` error, and JET scripts still emit a message that the extension service application was started. You must always refer to the system log messages to confirm that the device prevented a script from running.

Configure Start Options for Individual Scripts

To configure script start options for individual scripts:

1. Configure the threshold for system memory usage above which the device does not execute the specified script. Configure the threshold percentage (1 through 100) at the `file` *filename* hierarchy level for the specific script.

```
[edit event-options event-script file filename]
user@host# set start start-options mem-factor percentage
```

```
[edit system extensions extension-service application file filename]
user@host# set start start-options mem-factor percentage
```

```
[edit system scripts op file filename]
user@host# set start start-options mem-factor percentage
```

```
[edit system scripts snmp file filename]
user@host# set start start-options mem-factor percentage
```

2. Commit the configuration.

```
[edit]
user@host# commit
```

Configure Start Option for All Scripts of a Given Type

To configure script start options for all scripts of a given type:

1. Configure the threshold for system memory usage above which the device does not execute any script of the specified type. Configure the threshold percentage (1 through 100) at the configuration hierarchy level for that script type.

```
[edit event-options event-script]
user@host# set start start-options mem-factor percentage
```

```
[edit system extensions extension-service application]
user@host# set start start-options mem-factor percentage
```

```
[edit system scripts op]
user@host# set start start-options mem-factor percentage
```

```
[edit system scripts snmp]
user@host# set start start-options mem-factor percentage
```

2. Commit the configuration.

```
[edit]
user@host# commit
```

RELATED DOCUMENTATION

start (Scripts)

Understanding Limits on Executed Event Policies and Memory Allocation for Scripts

You can configure limits on the maximum number of concurrently running event policies and the maximum amount of memory allocated for the data segment for specific types of scripts. Depending on the device and its function in the network, it might be necessary to configure larger or smaller limits on the number of event policies that can execute concurrently and the maximum amount of memory allocated to scripts. You might configure smaller limits on critical devices to ensure that priority processes are not adversely impacted, and that the device can perform all necessary functions in the network.

Additionally, during normal device operation, you might want to allocate disproportionate amounts of memory to different script types. A device might have a particular type of script that plays a vital role in its operation and requires a specific amount of memory to ensure proper execution. For example, when committing a configuration that is inspected by one or more commit scripts, you might need to increase the amount of memory allocated to commit scripts to accommodate the processing of large configurations.

The default number of event policies that can run concurrently in the system is 15. If the system is running the maximum number of event policies, the system ignores any triggered event policy until such time that another policy finishes. The system logs the EVENTD_POLICY_LIMIT_EXCEEDED message for any triggered event policies that were not executed.

If you do not configure any limits, the default amount of memory that a device allocates to the data segment portion of an executed script depends on the operating system and release. Table 41 on page 503 outlines the default memory allocation. If a script requires more memory during execution than the set maximum limit, the script exits.

Table 41: Default Memory Allocation for Scripts

OS	Release	Default
Junos OS	–	Half of the total available memory of the system, up to a maximum value of 128 MB.
Junos OS Evolved	21.3 and earlier	128 MB
	21.4R1 and later	1024 MB

To set the maximum number of event policies that can run concurrently on a device:

- Configure the `max-policies` statement at the `[edit event-options]` hierarchy level, and specify the number of policies. You can configure a maximum of 0 through 20 policies.

```
[edit]
user@host# set event-options max-policies number
```

For example:

```
[edit]
user@host# set event-options max-policies 18
```

To set the maximum memory allocated to the data segment for scripts of a given type, configure the `max-datasize` *size* statement under the hierarchy appropriate for that script type, where *size* is the memory in bytes. To specify the memory in kilobytes, megabytes, or gigabytes, append `k`, `m`, or `g`, respectively, to the size. The allowed values vary by operating system, release, and script type. For more information see, *max-datasize*.

```
[edit]
event-options {
    event-script {
        max-datasize size;
    }
}
system {
    scripts {
        commit {
            max-datasize size;
        }
        op {
            max-datasize size;
        }
        snmp {
            max-datasize size;
        }
        translation {
            max-datasize size;
        }
    }
}
```

When you configure the `max-datasize` statement and execute a script, the device sets the maximum memory limit for that script to the configured value irrespective of the total memory available on the system at the time of execution. If the script exceeds the maximum memory limit during execution, it exits gracefully.

RELATED DOCUMENTATION

Example: Configure Limits on Executed Event Policies and Memory Allocation for Scripts

IN THIS SECTION

On Junos devices, you can configure the maximum number of event policies that can run concurrently on the system and the maximum amount of memory allocated for the data segment for scripts of a given type. This example configures a limit on the number of event polices that the device can execute concurrently and also configures maximum memory limits for executed commit, event, op, and SNMP scripts.

Requirements

A device running Junos OS Release 12.3 or later.

Overview

This example configures the device to limit the number of event policies that can run simultaneously to a maximum of 12 policies. Additionally, the example configures each script type with a maximum amount of memory that the system can allocate to the data segment portion of a script of that type. The device is configured to allocate 192 MB for each executed commit script and event script and 100 MB for each executed op and SNMP script.

Configuration

IN THIS SECTION

CLI Quick Configuration

To quickly configure this example, copy the following commands, paste them in a text file, remove any line breaks, change any details necessary to match your network configuration, and then copy and paste the commands into the CLI at the `[edit]` hierarchy level:

```
set system scripts commit max-datasize 192m
set system scripts op max-datasize 100m
set system scripts snmp max-datasize 100m
set event-options max-policies 12
set event-options event-script max-datasize 192m
```

Procedure

Step-by-Step Procedure

1. Configure the maximum number of event policies that can execute concurrently.

```
[edit]
user@host# set event-options max-policies 12
```

2. Configure the maximum memory allocated for the data segment for each executed commit script.

```
[edit]
user@host# set system scripts commit max-datasize 192m
```

3. Configure the maximum memory allocated for the data segment for each executed op script.

```
[edit]
user@host# set system scripts op max-datasize 100m
```

4. Configure the maximum memory allocated for the data segment for each executed SNMP script.

```
[edit]
user@host# set system scripts snmp max-datasize 100m
```

5. Configure the maximum memory allocated for the data segment for each executed event script.

```
[edit]
user@host# set event-options event-script max-datasize 192m
```

6. Commit the configuration.

```
[edit]
user@host# commit
```

Results

```
[edit]
event-options {
    event-script {
        max-datasize 192m;
    }
    max-policies 12;
}
system {
    scripts {
        commit {
            max-datasize 192m;
        }
        op {
            max-datasize 100m;
        }
        snmp {
```

```
            max-datasize 100m;
        }
    }
}
```

Verification

IN THIS SECTION

Confirm that the configuration is working properly.

Verifying the Limit on Concurrently Executing Event Policies

Purpose

If the system is running the maximum number of event policies, the system ignores any triggered event policy until such time that another policy finishes. The system logs the EVENTD_POLICY_LIMIT_EXCEEDED messsage for any triggered event policies that were not executed. By default, system log messages are recorded in the **messages** log file.

Action

Review the configured log file to verify whether any policies were barred from execution, because the maximum limit was reached. You can narrow the output to include only the relevant error messages by appending `| match EVENTD_POLICY_LIMIT_EXCEEDED`.

```
user@R1> show log messages | match EVENTD_POLICY_LIMIT_EXCEEDED
Jun 11 17:02:42  R1 eventd[1177]: EVENTD_POLICY_LIMIT_EXCEEDED: Unable to execute policy 'raise-
trap' because current number of policies (12) exceeds system limit (12)
[output omitted]
```

RELATED DOCUMENTATION

Understanding Limits on Executed Event Policies and Memory Allocation for Scripts | 502

max-datasize

max-policies

Dampen Script Execution

The Junos software automation feature enables you to create commit, op, event, and SNMP scripts to automate Junos devices. These scripts can be CPU-intensive, potentially impacting other critical software processes such as the routing protocol process (rpd). You can configure the device to dampen or slow down the execution of SLAX and XSLT commit, op, and event scripts. Although the scripts are processed more slowly, enabling script dampening helps to ensure that the other critical software processes can function normally.

NOTE: Script dampening is only supported for SLAX and XSLT scripts. Junos devices do not support script dampening for Python automation scripts.

The script dampening feature does not work for certain CPU-intensive remote procedure calls (*RPC*s), including the following:

- Executing the following RPC might cause the software process to consume numerous CPU cycles. This cannot be avoided as it is outside the control of the cscript process and up to the software process to optimize.

```
var $cmd = <command> "show snmp mib walk .1";
var $out = jcs:invoke($cmd);
```

- The cscript process might consume numerous CPU cycles for this particular RPC, as this involves parsing of the XML configuration returned by the management process (mgd). This cannot be avoided since a single line in the script triggers the processing.

```
var $config = jcs:invoke("get-configuration");
```

To enable script dampening:

1. Include the `dampen` statement.

```
dampen;
```

The `dampen` statement and the script dampening options configured under the `dampen-options` statement can be configured at various hierarchy levels, depending on the type of scripts you want to dampen and whether or not you want to apply dampening to specific scripts or all scripts.

- You can dampen all event scripts by configuring the `dampen` statement at the `[edit event-options event-script]` hierarchy level or dampen a specific event script by configuring the `dampen` statement at the `[edit event-options event-script file` *`filename`*`]` hierarchy level.

- You can dampen all op scripts by configuring the `dampen` statement at the `[edit system scripts op]` hierarchy level or dampen a specific op script by configuring the `dampen` statement at the `[edit system scripts op file` *`filename`*`]` hierarchy level.

  NOTE: Script dampening is only enforced for op scripts that are local to the device. If you execute an op script from a remote location using the `op url` command, Junos OS does not dampen the script.

- You can dampen all commit scripts by configuring the `dampen` statement at the `[edit system scripts commit]` hierarchy level. You cannot dampen a specific commit script. This is a limitation of the cscript process which runs a single commit script at a time.

2. (Optional) Modify the behavior of the script dampening feature by configuring the following dampening options:

   - *cpu-factor*—Script dampening is initiated when CPU use exceeds the value specified using this statement.

   - *line-interval*—Specifies the number of lines of script to execute before pausing.

   - *time-interval*—Specifies the time that script execution is paused.

   To configure the script dampening options, configure the following statements. If you configure the `dampen` statement and do not configure values for these statements, the defaults are used. You can configure the `dampen-options` statements at all of the hierarchy levels at which you can configure the `dampen` statement.

   If you configure the `dampen-options` statements at both a global hierarchy level (for example, at the `[edit event-options event-script]` hierarchy level) or for a specific script (for example, at the `[edit event-options event-script file` *`filename`*`]` hierarchy level), the file hierarchy level configuration takes precedence.

```
dampen-options {
    cpu-factor cpu-factor;
    line-interval line-interval;
    time-interval microseconds;
}
```

3. (Optional) On devices running Junos OS, configure the `traceoptions` statement to help debug issues related to the script dampening feature.

   - For commit scripts, include the `events` flag in the *traceoptions* statement configuration.

     ```
     [edit system scripts commit traceoptions]
     flag events;
     ```

   - For event scripts, include the `events` flag in the *traceoptions* statement configuration.

     ```
     [edit event-options event-script traceoptions]
     flag events;
     ```

   - For op scripts, include the `events` flag in the *traceoptions* statement configuration.

     ```
     [edit system scripts op traceoptions]
     flag events;
     ```

   The following traceoptions file output is an example of the trace message logged when a script is paused by the script dampening feature.

   ```
   Dec 11 21:40:40 cscript: paused for 100 microseconds
   ```

RELATED DOCUMENTATION

dampen

dampen-options

CHAPTER 19

Synchronize Scripts Between Routing Engines

IN THIS CHAPTER

Understanding Script Synchronization Between Routing Engines

Starting in Junos OS Release 13.2, you can manually synchronize commit, event, lib, op, and SNMP scripts between Routing Engines on a device running Junos OS or configure the device to automatically synchronize scripts between Routing Engines when you commit and synchronize the configuration. When invoked, the device synchronizes the scripts from the Routing Engine on which you execute the request (the requesting Routing Engine) to the other Routing Engine (the responding Routing Engine).

In operational mode, you can manually synchronize scripts from the requesting Routing Engine to the responding Routing Engine using the `request system scripts synchronize` command. The command enables you to customize the scope of the synchronization. You can synchronize a single script, all scripts of a specific type, or all scripts on the device. You also have the option to synchronize scripts based on filename or on the timestamp of the file.

In configuration mode, you have the option to synchronize all scripts when you commit and synchronize the configuration. To synchronize scripts on a per-commit basis, use the `commit synchronize scripts` command when you commit and synchronize the configuration. Alternatively, you can configure the device to automatically synchronize scripts from the requesting Routing Engine to the responding Routing Engine every time you issue a `commit synchronize` command. To ensure that scripts are copied from the requesting Routing Engine to the responding Routing Engine during a `commit synchronize` operation, configure the `synchronize` statement at the `[edit system scripts]` hierarchy level.

When you synchronize the configuration and scripts, the device:

1. Performs a commit check on the requesting Routing Engine

2. Synchronizes scripts to the responding Routing Engine

3. Synchronizes the configuration to the responding Routing Engine

4. Performs a commit check on the responding Routing Engine

5. Commits the configuration on the responding Routing Engine

6. Commits the configuration on the requesting Routing Engine

This process ensures that any commit scripts that are required for a successful commit operation are present on the responding Routing Engine before committing the configuration. If the commit check operation fails for the requesting Routing Engine, the process stops, and the scripts are not copied to the responding Routing Engine. If the commit check or commit operation fails for the responding Routing Engine, the scripts are still synchronized since the synchronization occurs prior to the commit check operation on the responding Routing Engine.

When synchronizing scripts, the device running Junos OS determines the script source and destination directories based on whether the `load-scripts-from-flash` statement is present in the configuration for each Routing Engine. If the `load-scripts-from-flash` statement is configured for the requesting Routing Engine, the device synchronizes the scripts that are in flash memory. Otherwise, the device synchronizes the scripts that are on the hard disk. If the `load-scripts-from-flash` statement is present in the final configuration for the responding Routing Engine, the scripts are synchronized to flash memory. Otherwise, the scripts are synchronized to the hard disk. The device synchronizes a script regardless of whether it is enabled in the configuration or has been updated since the last synchronization.

The `request system scripts refresh-from` *operational mode command* enables you to manually refresh a single script from a remote URL. Starting in Junos OS Release 13.2, you can synchronize the updated script to the other Routing Engine at the same time by including the `sync` option when you execute the command. When you execute the command, if the `load-scripts-from-flash` statement is configured on the Routing Engine, the script is refreshed in flash memory. Otherwise, the script is refreshed on the hard disk.

RELATED DOCUMENTATION

Synchronize Scripts Between Routing Engines | 514

Example: Synchronize Scripts Between Routing Engines | 518

synchronize

request system scripts synchronize

request system scripts refresh-from

Synchronize Scripts Between Routing Engines

IN THIS SECTION

Starting in Junos OS Release 13.2, you can manually synchronize commit, event, lib, op, and SNMP scripts between Routing Engines on a device running Junos OS or configure the device to automatically synchronize scripts between Routing Engines when you commit and synchronize the configuration.

If the `load-scripts-from-flash` statement is configured for the requesting Routing Engine, the device synchronizes the scripts that are in flash memory. Otherwise, the device synchronizes the scripts that are on the hard disk. If the `load-scripts-from-flash` statement is present in the final configuration for the responding Routing Engine, the scripts are synchronized to flash memory. Otherwise, the scripts are synchronized to the hard disk. The device synchronizes a script regardless of whether it is enabled in the configuration or has been updated since the last synchronization.

The following sections outline the different methods for synchronizing scripts:

Configuring Script Synchronization Between Routing Engines for Commit Synchronize Operations

You can configure a device running Junos OS to synchronize all commit, event, lib, op, and SNMP scripts from the requesting Routing Engine to the other Routing Engine every time you issue the `commit synchronize` command to commit and synchronize the configuration.

To automatically synchronize scripts between Routing Engines during a `commit synchronize` operation:

1. Configure the `synchronize` statement at the `[edit system scripts]` hierarchy level.

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

2. Commit and synchronize the configuration.

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

When you issue the first and subsequent `commit synchronize` commands, the device performs a commit check on the requesting Routing Engine, synchronizes all scripts to the other Routing Engine, synchronizes, performs a commit check, and commits the configuration on the responding Routing Engine, and finally commits the configuration on the requesting Routing Engine. If the commit check operation fails for the requesting Routing Engine, the process stops, and the scripts are not copied to the responding Routing Engine. If the commit check or commit operation fails for the responding Routing Engine, the scripts are still synchronized since the synchronization occurs prior to the commit check operation on the responding Routing Engine.

Configuring the `synchronize` statement causes the device to synchronize all scripts even if they have not been updated since the last synchronization. If the device has a large number of scripts that are infrequently updated, it might be more suitable to synchronize scripts either manually using the `request system scripts synchronize` operational mode command or on a per-commit basis using the `commit synchronize scripts` command.

Synchronizing Scripts Between Routing Engines on a Per-Commit Basis

You can synchronize all commit, event, lib, op, and SNMP scripts from the requesting Routing Engine to the other Routing Engine on a device running Junos OS on a per-commit basis using the `commit synchronize scripts` command when you commit and synchronize the configuration. This is an alternative to configuring the device to synchronize scripts every time you execute a `commit synchronize` operation.

To synchronize scripts between Routing Engines on a per-commit basis:

1. Make all necessary changes to the configuration.
2. Issue the `commit synchronize scripts` command.

```
[edit]
user@host# commit synchronize scripts
```

When you issue the `commit synchronize scripts` command, the device performs a commit check on the requesting Routing Engine, synchronizes all scripts to the other Routing Engine, synchronizes, performs a commit check, and commits the configuration on the responding Routing Engine, and finally commits the configuration on the requesting Routing Engine. If the commit check operation fails for the requesting Routing Engine, the process stops, and the scripts are not copied to the responding Routing Engine. If the commit check or commit operation fails for the responding Routing Engine, the scripts are still synchronized since the synchronization occurs prior to the commit check operation on the responding Routing Engine.

Synchronizing Scripts Between Routing Engines from Operational Mode

You can manually synchronize scripts from the requesting Routing Engine to the other Routing Engine on a device running Junos OS by using the `request system scripts synchronize` operational mode command. You can synchronize a single script, all scripts of a specific type, or all scripts on the device. You also have the option to synchronize scripts based on the filename or on the timestamp of the file.

To manually synchronize scripts between Routing Engines, issue the `request system scripts synchronize` command with the desired options.

```
user@host> request system scripts synchronize (all | commit | event | lib | op | snmp)
<file filename> <newer-than time>
```

Specify `all` to synchronize all scripts present on the requesting Routing Engine to the responding Routing Engine. Specify `commit`, `event`, `lib`, `op`, or `snmp` to synchronize all scripts of the given type to the other Routing Engine. Include the `file` option or the `newer-than` option to narrow the scope to only synchronize scripts with the specified filename or date criteria. The format for the `newer-than` argument is YYYY-MM-DD.HH:MM:SS.

For example, the following command synchronizes all commit, event, lib, op, and SNMP scripts that have a timestamp newer than 2012-05-15:

```
user@host> request system scripts synchronize all newer-than 2012-05-15
```

The following command synchronizes a single op script with the filename **vpn-info.slax**.

```
user@host> request system scripts synchronize op file vpn-info.slax
```

A synchronize operation might fail if, for example, you request to synchronize a script that does not exist or if the responding Routing Engine cannot handle the request at that time, because it is performing other CPU-intensive operations. If the synchronize operation fails, the device generates an error message.

The following command requests to synchronize a single event script, but the script does not exist in the event scripts directory, so the device issues an error.

```
user@host> request system scripts synchronize event file nonexistent-file.slax
```

```
error: Invalid directory: No such file or directory
warning: No script will be pushed to other RE
```

The following command requests to synchronize a single event script, but the responding Routing Engine does not have the resources to perform the synchronization, so the device issues an error. The device also logs a UI_SCRIPTS_COPY_FAILED error in the system log file with a severity level of error.

```
user@host> request system scripts synchronize event file ospf-neighbor.slax
```

```
error: Unable to copy scripts to re: re1
```

Synchronizing a Script Between Routing Engines After a Refresh

You can manually refresh a single script from a remote URL and synchronize the updated script to the other Routing Engine on a device running Junos OS by using the `request system scripts refresh-from` operational mode command with the `sync` option.

To manually refresh a single script from a remote source and then synchronize the script to the other Routing Engine, issue the `request system scripts refresh-from` command with the `sync` option, and specify the script type, filename, and remote URL. Specify the URL as an HTTP URL, FTP URL, or secure copy (scp)-style remote file specification.

```
user@host> request system scripts refresh-from (commit | event | op | snmp) file filename
url url sync
```

The system uses the script type to determine the directory on the device where the script resides. If the `load-scripts-from-flash` statement is present in the configuration for the Routing Engine, the system refreshes the script in flash memory. Otherwise, the system refreshes the script on the hard disk.

RELATED DOCUMENTATION

Understanding Script Synchronization Between Routing Engines | 512

Example: Synchronize Scripts Between Routing Engines

IN THIS SECTION

This example shows how to configure a device with dual Routing Engines running Junos OS to synchronize all commit, event, lib, op, and SNMP scripts between Routing Engines every time you execute a `commit synchronize` operation.

Requirements

A routing, switching, or security device with dual Routing Engines running Junos OS Release 13.2 (15.1 for SNMP scripts) or later is required.

Overview

In this example, you configure a device with dual Routing Engines running Junos OS to synchronize all commit, event, lib, op, and SNMP scripts from the requesting Routing Engine to the other Routing Engine whenever you execute a `commit synchronize` command to commit and synchronize the configuration. When configured, the device synchronizes all scripts regardless of whether they are enabled in the configuration.

In this example, the `load-scripts-from-flash` statement is not configured on the requesting Routing Engine. Thus, the device synchronizes the scripts that are on the hard disk of the requesting Routing Engine to the hard disk of the responding Routing Engine.

NOTE: On the hard disk, scripts are stored under the **/var/db/scripts** directory in the subdirectory appropriate to the script type. In flash memory, scripts are stored under the **/config/scripts** directory in the subdirectory appropriate to the script type. EX Series switches use the default directory `/config/db/scripts`.

Configuration

IN THIS SECTION

CLI Quick Configuration

To quickly configure this example, copy the following command, paste it in a text file, and then copy and paste the command into the CLI at the `[edit]` hierarchy level.

```
set system scripts synchronize
```

Configuring Script Synchronization for Commit Synchronize Operations

Step-by-Step Procedure

To automatically synchronize scripts between Routing Engines during a `commit synchronize` operation:

1. Configure the `synchronize` statement at the `[edit system scripts]` hierarchy level.

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

2. Commit and synchronize the configuration.

```
[edit system scripts]
user@host# commit synchronize
re0:
configuration check succeeds
re1:
commit complete
re0:
commit complete
```

When you issue the first and subsequent `commit synchronize` commands, the device performs a commit check on the requesting Routing Engine, synchronizes all scripts to the other Routing Engine, synchronizes, performs a commit check, and commits the configuration on the responding Routing Engine, and finally commits the configuration on the requesting Routing Engine.

Results

The resulting configuration is:

```
system {
    scripts {
        synchronize;
    }
}
```

Verification

IN THIS SECTION

Confirm that the configuration is working properly and the synchronization is successful.

Verifying Script Synchronization

Purpose

Verify that the scripts present on the requesting Routing Engine are synchronized to the other Routing Engine.

In this example, the `load-scripts-from-flash` statement is not configured for the requesting Routing Engine. Therefore, the device synchronizes scripts from the **/var/db/scripts** directory on the requesting Routing Engine to the **/var/db/scripts** directory on the responding Routing Engine.

Action

Use the `file list` operational mode command to view the files in the **/var/db/scripts** directory on each Routing Engine.

1. On the requesting Routing Engine, list the files under the **/var/db/scripts/** directory.

```
user@host> file list /var/db/scripts/* detail
/var/db/scripts/commit:
-rw-r--r--  1 remote wheel      1014 Jul 17 16:18 vpn-commit.slax

/var/db/scripts/event:

/var/db/scripts/lib:

/var/db/scripts/op:
-rw-r--r--  1 root  wheel     11485 Sep 21  2010 jcs-load-config-op.slax
```

2. Log in to the responding Routing Engine, and verify that the files are synchronized.

```
user@host> request routing-engine login other-routing-engine
user@host1> file list /var/db/scripts/* detail
/var/db/scripts/commit:
-rw-r--r--  1 remote wheel      1014 Jul 17 16:18 vpn-commit.slax

/var/db/scripts/event:

/var/db/scripts/lib:
```

```
/var/db/scripts/op:
-rw-r--r--  1 root  wheel     11485 Sep 21  2010 jcs-load-config-op.slax
```

Meaning

The scripts present on each Routing Engine are identical indicating that the device successfully synchronized the scripts from the requesting Routing Engine to the responding Routing Engine.

Troubleshooting

IN THIS SECTION

Troubleshooting Script Synchronization Failure

Problem

The device does not synchronize the scripts present on the requesting Routing Engine to the other Routing Engine.

Solution

Verify the following:

- You configured the `synchronize` statement at the `[edit system scripts]` hierarchy level.
- You are viewing the correct directories on each Routing Engine.

  If the `load-scripts-from-flash` statement is configured for the requesting Routing Engine, the device synchronizes scripts from flash memory on the requesting Routing Engine to flash memory on the responding Routing Engine.
- You executed a `commit synchronize` command when committing the configuration.

  The device does not synchronize scripts for a `commit` operation, only for a `commit synchronize` operation.
- The commit check and commit operations for the requesting Routing Engine are successful.

  If the commit check operation for the requesting Routing Engine fails, the process stops, and the scripts are not copied to the responding Routing Engine.

RELATED DOCUMENTATION

Understanding Script Synchronization Between Routing Engines | 512

Synchronize Scripts Between Routing Engines | 514

synchronize

CHAPTER 20

Convert Scripts Between SLAX and XSLT

IN THIS CHAPTER

Convert Scripts Between SLAX and XSLT

IN THIS SECTION

SLAX is a C-like alternative syntax to XSLT and can be viewed as a preprocessor for XSLT. Before Junos OS invokes the XSLT processor, the software converts any SLAX constructs in the script (such as `if/else if/else`) to equivalent XSLT constructs (such as `<xsl:choose>` and `<xsl:if>`). For more information about SLAX, see "SLAX Overview" on page 83.

You can use the `request system scripts convert` operational mode command to convert a script or partial script input written in SLAX or XSLT into the alternate language. Users familiar with C and PERL can convert existing XSLT scripts to SLAX 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

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. Starting in Junos OS Release 12.2, you can also include the `partial` option to convert partial script input.

The command syntax is:

```
user@host> request system scripts convert slax-to-xslt source source/filename destination
destination/<filename> <partial>
```

The following three examples convert a script from SLAX to XSLT 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 filename is **SLAX-Conversion-Temp** or **slax-temp** depending on the Junos OS release, with a randomly generated, five-character, alpha-numeric extension.

```
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/filename destination
destination/<filename> <partial> <version (1.0 | 1.1)>
```

To convert partial script input, include the `partial` option in the command. The `version` option specifies the SLAX version that will be listed in the version statement of the generated script. Specify the version as either 1.0 or 1.1. The default is 1.1. The `partial` and `version` options are supported starting in Junos OS Release 12.2.

The following three examples convert a script from XSLT to SLAX 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 version 1.0
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
```

The SLAX script boilerplate lists the specified SLAX version. In this example, the version is 1.0.

```
user@host> file show /var/db/scripts/op/script1.slax
/* Machine Crafted with Care (tm) by slaxWriter */

version 1.0;
...
```

If you specify only the destination directory and do not specify a destination filename, the generated filename is **SLAX-Conversion-Temp** or **slax-temp** depending on the Junos OS release, with a randomly generated, five-character, alpha-numeric extension.

```
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-temp.Vosnd
script1.xsl
```

RELATED DOCUMENTATION

SLAX Overview | 83

8

PART

Commit Scripts

CHAPTER 21

Commit Scripts Overview

IN THIS CHAPTER

Commit Script Overview

IN THIS SECTION

Understanding Commit Scripts

You can use Junos OS commit scripts to customize the validation process of your configurations in accordance with your own practices and policies and enforce custom configuration rules during the commit process. When you commit a *candidate configuration*, it is inspected by each active *commit script*. If a configuration violates your custom rules, the script can instruct Junos OS to take appropriate action. A commit script can:

- Generate and display custom warning messages to the user.
- Generate and log custom system log (syslog) messages.
- Change the configuration to conform to your 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 OS configuration statements and operational mode commands.

You can write commit scripts in Python, Extensible Stylesheet Language Transformations (*XSLT*), or Stylesheet Language Alternative syntaX (SLAX). The Junos XML API defines an XML equivalent for all statements in the Junos configuration hierarchy. Commit scripts use *XML Path Language* (XPath) to locate the configuration objects to inspect, and they use automation script constructs to specify the actions to perform on the configuration objects. The actions can generate messages or change the configuration.

Additionally, you can create *macros*, which enable you to use custom configuration syntax that simplifies the task of configuring Junos devices. 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 OS configuration statements that execute your intended operational impact.

To view the device's current configuration in XML, issue the `show configuration | display xml` command in CLI operational mode. To view your configuration in commit-script-style XML, issue the `show configuration | display commit-scripts view` command. Commit-script-style XML view displays the configuration in the format that would be input to a commit script.

Benefits of Commit Scripts

Commit scripts provide the following benefits:

- Enable the enforcement of custom configuration rules.
- Improve network reliability and uptime by minimizing human error.
- Automatically correct configuration mistakes during a commit operation.
- Abstract and simplify complex configurations.
- Enforce scaling limits for critical settings.

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 are not 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 an MTU size less than 4 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 the configuration never enables Level 1 IS-IS routers.

When a *candidate configuration* does not adhere to your design rules, a commit script can instruct Junos OS to generate custom warnings, record system log messages, or generate 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 you to enable MPLS on every interface where you enable the ISO family of protocols. At commit time, a commit script inspects the configuration and issues an error if the configuration doesn't meet this requirement. 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 the configuration to enable MPLS on all interfaces. The commit script can also generate a system log message, indicating that the script took corrective action on the configuration.

Another option is to define a macro that enables ISO protocols and MPLS when you apply the macro 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 can enable MPLS on ISO-enabled interfaces without displaying the corresponding configuration statements in the candidate configuration.

NOTE: Commit scripts generate transient changes in the *checkout configuration* but not in the candidate configuration. The checkout configuration is the configuration database that the system checks for standard Junos OS syntax just before a configuration becomes active. This means transient changes are not saved in the configuration if you delete or deactivate the associated commit script. The `show configuration | display commit-scripts` command displays all the statements that are in the configuration, including statements that are generated by transient changes. For more information, see "Overview of Generating Persistent or Transient Configuration Changes Using Commit Scripts" on page 598.

How Commit Scripts Work

IN THIS SECTION

Commit scripts contain instructions that enforce custom configuration rules. When you commit the configuration, the system invokes the commit scripts during the commit process before the standard Junos OS validity checks are performed. You enable commit scripts by configuring the names of one or more commit script files at the `[edit system scripts commit]` hierarchy level. These files must be added to the appropriate commit script directory on the device. For more information, see "Store and Enable Junos Automation Scripts" on page 448

When you perform a commit operation, Junos OS executes each commit script in turn, passing the information in the post-inheritance *candidate configuration* to the script. The script inspects the configuration, performs the necessary tests and validations, and generates a set of instructions for performing certain actions. After all commit scripts are executed, Junos OS then processes all of the scripts' instructions. If a commit script instruction does not halt the commit process, then Junos OS applies all the commit script changes and performs its final inspection of the checkout configuration.

NOTE: You might need to increase the amount of memory allocated to commit scripts to accommodate the processing of large configurations. To increase the maximum memory allocated for each executed commit script, configure the `max-datasize` *`size`* statement with an appropriate memory limit in bytes at the `[edit system scripts commit]` hierarchy level before committing the configuration.

Commit script actions can include generating error, warning, and system log messages. If a script generate an error, 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.

Commit scripts cannot make configuration changes to protected statements or within protected hierarchies. If a commit script attempts to modify or delete a protected statement or hierarchy, Junos OS issues a warning that the change cannot be made. Failure to modify a protected configuration element does not halt the commit script or the commit process.

The following sections discuss several important concepts related to the commit script input and output:

Commit Script Input

The input for a commit script is the post-inheritance candidate configuration in Junos XML API format. The term *post-inheritance* means that all configuration group values have been inherited by their targets in the candidate configuration and that the inactive portions of the configuration have been removed. For more information about configuration groups, see the CLI User Guide.

When you issue the `commit` command, 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 post-inheritance candidate configuration in the CLI, 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
```

Commit Script Output

During the commit process, enabled commit scripts are executed sequentially, and the commit script output, or instruction set, is provided to Junos OS. After all commit scripts have been executed, Junos OS then processes all of the scripts' instructions.

Commit script actions can include generating warning, error, and system log messages, and making persistent and transient changes to the configuration. Table 42 on page 535 briefly outlines the various elements, templates, and functions that commit scripts can use to instruct Junos OS to perform various actions during the commit process. In some cases, there are multiple ways to perform the same action. Because SLAX and XSLT scripts return a result tree, output elements like `<syslog><message>` that are present in SLAX and XSLT scripts are added directly into the result tree.

Table 42: Commit Scripts Actions and Output

Commit Script Output	SLAX / XSLT	Python
Generate a warning message to the committing user.	`<xnm:warning>`	jcs.emit_warning()
Generate an error message and cause the commit operation to fail.	`<xnm:error>`	jcs.emit_error()
Generate a system log message.	jcs:syslog() `<syslog><message>`	jcs.syslog()
Generate a persistent change to the configuration.	`<change>`	emit_change(*content*, 'change', *format*)
Generate a transient change to the configuration.	`<transient-change>`	emit_change(*content*, 'transient-change', *format*)

Table 42: Commit Scripts Actions and Output *(Continued)*

Commit Script Output	SLAX / XSLT	Python

Generate a persistent change relative to the current *context node* as defined by an *XPath* expression.	**XSLT**
	`<xsl:call-template name="jcs:emit-change">`
	`    <xsl:with-param name="content">`
	SLAX
	`call jcs:emit-change() {`
	`    with $content = {    }`
	`}`
		–

Generate a transient change relative to the current context node as defined by an XPath expression.	**XSLT**
	`<xsl:call-template name="jcs:emit-change">`
	`    <xsl:with-param name="tag" select="'transient-change'"/>`
	`    <xsl:with-param name="content">`
	SLAX
	`call jcs:emit-change() {`
	`    with $tag = "transient-change";`
	`    with $content = {    }`
	`}`
		–

Table 42: Commit Scripts Actions and Output *(Continued)*

Commit Script Output	SLAX / XSLT	Python

Generate 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.	**XSLT**
	<xsl:call-template name="jcs:emit-change">
	    <xsl:with-param name="message">
	        <xsl:text>
	SLAX
	call jcs:emit-change() {
	    with $message = {
	        expr "*message*";
	    }
	}
		jcs.emit_warning()

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.

Commit Scripts and the Junos OS Commit Model

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.

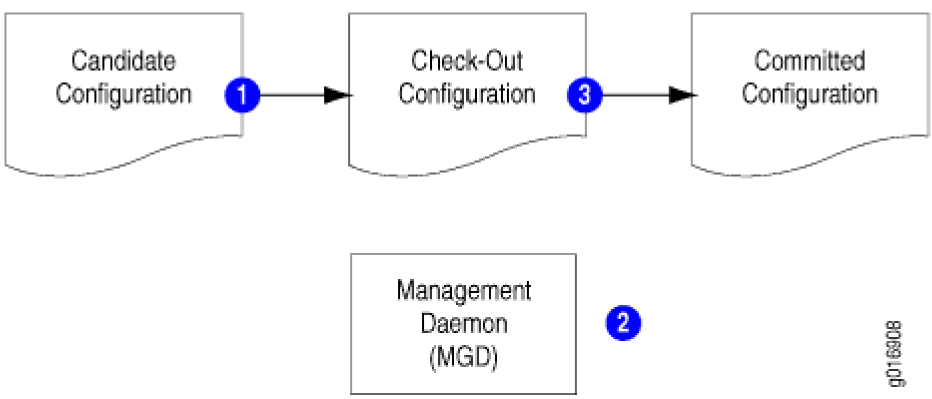
When you commit a set of changes in the candidate configuration, two methods are used to forward these changes to the current configuration:

- 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 OS validation rules. If the configuration file is valid, it becomes the current active configuration. Figure 4 on page 538 and the accompanying discussion explain how the standard commit model works.

Figure 4: Standard Commit Model



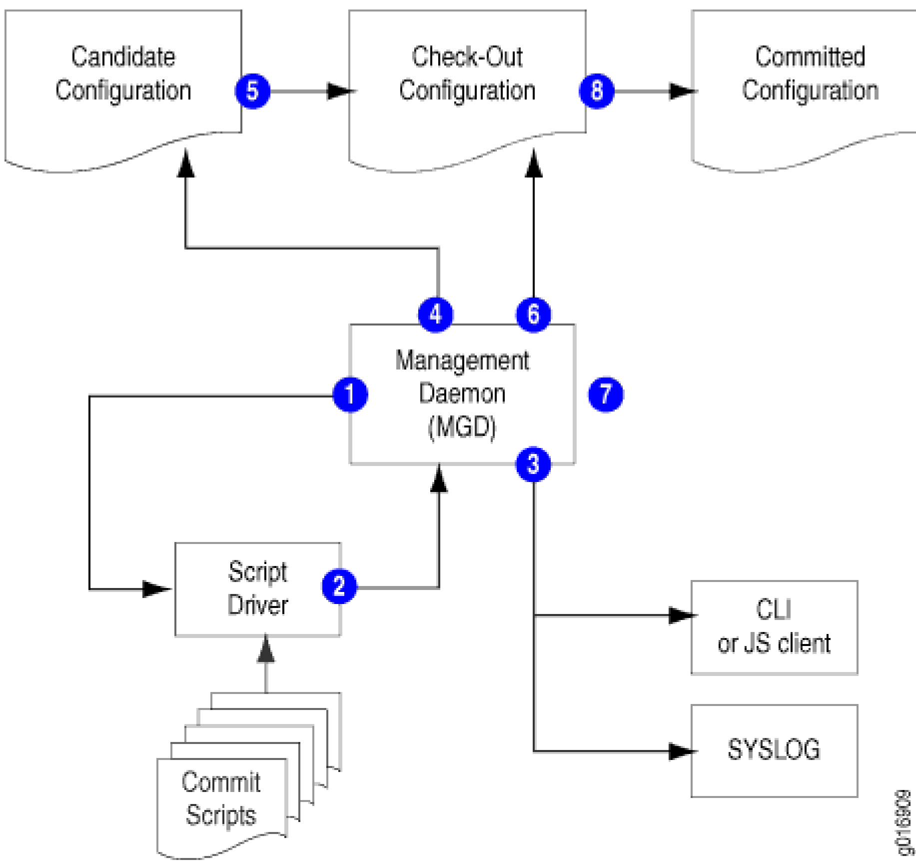
In the standard commit model, the software performs the following steps:

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 *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 or client application, and write information to the system log as required. After processing the action file, the mgd process performs the standard Junos OS validation. Figure 5 on page 539 and the accompanying discussion explain this process.

Figure 5: Commit Model with Commit Scripts Added



In the commit script model, 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 actions are collected in an action file.

3. The mgd process performs the following actions for commit script error, warning, and system log messages 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.
   - system log message—The mgd process forwards the message to the system log process.

4. If the action file includes any persistent changes, 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 changes, 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.

> **NOTE**: Commit scripts cannot make configuration changes to protected statements or within protected hierarchies. If a commit script attempts to modify or delete a protected statement or hierarchy, Junos OS issues a warning that the change cannot be made. Failure to modify a protected configuration element does not halt the commit script or the commit process.

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 550.

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 "Control the Execution of Commit Scripts During Commit Operations" on page 557), 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 Using Commit Scripts" on page 598.

RELATED DOCUMENTATION

protect

emit-change Template (SLAX and XSLT) and emit_change (Python) | **434**

How to Avoid Potential Conflicts When Using Multiple Commit Scripts | **550**

CHAPTER 22

Create and Execute Commit Scripts

IN THIS CHAPTER

Required Boilerplate for Commit Scripts

SUMMARY

Define the boilerplate for commit scripts.

Junos OS commit scripts can be written in Extensible Stylesheet Language Transformations (*XSLT*), Stylesheet Language Alternative syntaX (SLAX), or Python. Commit scripts must include the necessary boilerplate required for that script language for both basic script functionality as well as any optional functionality used within the script such as the Junos OS extension functions and named templates. This topic provides standard boilerplate that can be used in XSLT, SLAX, and Python commit scripts.

SLAX and XSLT 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.

XSLT Boilerplate for Commit Scripts

The XSLT commit script boilerplate is as follows:

```
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"?>
```

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 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.

```
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 "Understanding Named Templates in Junos OS Automation Scripts" on page 427.

```
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 XPath expressions in the script and begin XPath expressions with the top Junos OS hierarchy level. For more information, see "XPath Overview" on page 22.

```
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 SLAX commit script boilerplate is as follows:

```
version 1.2;
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
```

```
    */
}
```

Python Boilerplate for Commit Scripts

Python commit scripts do not have a required boilerplate, but they must import any objects that are used in the script. Python commit scripts can import the following:

- `Junos_Context` dictionary—Contains information about the script execution environment.
- `Junos_Configuration` object—Contains the post-inheritance candidate configuration.
- `jcs` library—Enables the script to use Junos OS extension functions and Junos OS named template functionality in the script.
- `jnpr.junos` module and classes—Enables the script to use Junos PyEZ.

For example:

```
from junos import Junos_Context
from junos import Junos_Configuration
from jnpr.junos import Device
import jcs

if __name__ == '__main__':
```

Python automation scripts do not need to include an interpreter directive line (`#!/usr/bin/env python`) at the start of the script. However, the program will still execute correctly if one is present.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337
Understanding Named Templates in Junos OS Automation Scripts | 427
Global Parameters and Variables in Junos OS Automation Scripts | 327

XML Syntax for Common Commit Script Tasks

Junos OS commit scripts can generate persistent or transient changes to the configuration during the commit process. A commit script can perform common configuration tasks by adding the appropriate attribute to a specific XML tag in the configuration data. Table 43 on page 546 summarizes the tasks and the syntax for each task.

Table 43: XML Syntax for Common Commit Script Tasks

Action	Syntax	Example
Add a data element	normal XML	<address> <name>192.168.1.1</name> </address>
Remove the inactive tag from a statement	active="active"	<address active="active"> <name>192.168.1.1/30</name> </address>
Delete a data element	delete="delete"	<address delete="delete"> <name>192.168.1.1/30</name> </address>
Add the inactive tag to a statement	inactive="inactive"	<address inactive="inactive"> <name>192.168.1.1/30</name> </address>
Insert a new ordered data element	insert="(before | after)" name="*reference-value*"	<address insert="before" name="192.168.1.5/30"> <name>192.168.1.1/30</name> </address>
Add the protect tag to a statement or node to prevent configuration changes to that element	protect="protect"	<address protect="protect""> <name>192.168.1.1/30</name> </address>

Table 43: XML Syntax for Common Commit Script Tasks *(Continued)*

Action	Syntax	Example
Rename a statement	rename="rename" name="*new-name*"	<address rename="rename" name="192.168.1.1/30"> <name>192.168.1.5/30</name> </address>
Replace a node or statement in the hierarchy	replace="replace"	<system> <services replace="replace"> [...] </services> </system>
Unprotect a statement or node in the hierarchy	unprotect="unprotect"	<address unprotect="unprotect"> <name>192.168.1.1/30</name> </address>
Annotate a configuration statement with a comment	<junos:comment>	<system> <junos:comment> /* added by *username* */ </junos:comment> <services> [...] </services> </system>

Design Considerations for Commit Scripts

After you have 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 using *XSLT*.

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>
```

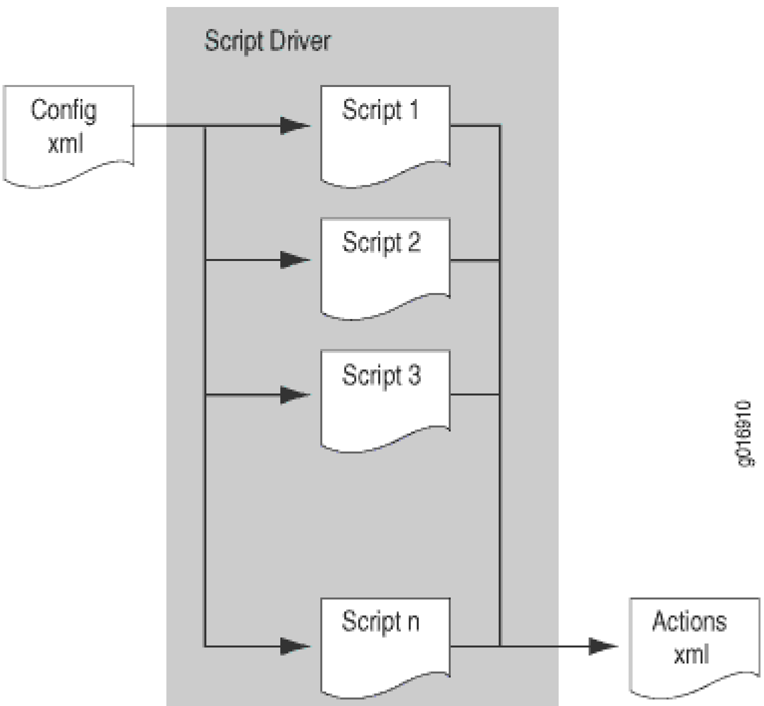
If you channelize a cstm1-0/1/0 interface into 17 E1 interfaces, the script causes the following error message to appear when you issue the `commit` command. (For more information about this example, see "Example: Limit the Number of E1 Interfaces" on page 742.)

```
[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
```

How to Avoid 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 6 on page 550.

Figure 6: 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 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.

As another example of a potential conflict between commit scripts, suppose that a commit script protects a hierarchy using the `protect` attribute. If a second commit script attempts to modify or delete the hierarchy or the statements within the hierarchy, Junos OS issues a warning during the commit process and prevents the configuration change.

Exercise care to ensure that you do not introduce conflicts between scripts like those described in the examples. As a method of checking for conflicts with persistent changes, you can issue two separate `commit` commands.

RELATED DOCUMENTATION

How Commit Scripts Work | 533

Line-by-Line Explanation of Sample Commit Scripts

IN THIS SECTION

Applying a Change to SONET/SDH Interfaces

The following XSLT 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 Using Commit Scripts" on page 598. For a SLAX version of this example, see "Example: Generating a Transient Change" on page 624.

```
<?xml version="1.0"?>
<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]">
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 542 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 XSLT 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: Control IS-IS and MPLS Interfaces" on page 710.

```
<?xml version="1.0"?>
<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/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>
```

```
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 542 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 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 "emit-change Template (SLAX and XSLT) and emit_change (Python) " on page 434.

```
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>
```

RELATED DOCUMENTATION

Example: Generating a Transient Change | 624

Control the Execution of Commit Scripts During Commit Operations

IN THIS SECTION

Commit scripts are stored on a device's hard disk 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 "Store and Enable Junos Automation Scripts" on page 448 and "Store Scripts in Flash Memory" on page 451.

A commit script is not actually executed during commit operations unless its filename is included at the `[edit system scripts commit file]` hierarchy level. When you configure the script filename, you must include the appropriate filename extension for SLAX (**.slax**) and Python (**.py**) scripts. XSLT scripts do not require a filename extension, but we strongly recommend that you append the **.xsl** extension for clarity. 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 enabled on the device, use the `show` command to display the files configured at the `[edit system scripts commit]` hierarchy level. To ensure that the enabled files are on the device, list the contents of the **/var/run/scripts/commit/** directory by using the `file list /var/run/scripts/commit` operational mode command.

Enabling Commit Scripts to Execute During Commit Operations

To configure a commit script to execute during a commit operation:

1. Ensure that the commit script is located in the correct directory:

   - **/var/db/scripts/commit** directory on the hard disk
   - **/config/scripts/commit** directory on the flash drive

   For more information about script storage location, see "Store and Enable Junos Automation Scripts" on page 448 and "Store Scripts in Flash Memory" on page 451.

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. For unsigned Python scripts, ensure that the following requirements are met:

   - File owner is either root or a user in the Junos OS `super-user` login class.
   - Only the file owner has write permission for the file.

- The `language python` or `language python3` statement is configured at the `[edit system scripts]` hierarchy level.

```
[edit system scripts]
user@host# set language (python | python3)
```

> **NOTE**: Starting in Junos OS Release 16.1R3, Python scripts can be owned by either root or a user in the Junos OS super-user login class. In Junos OS Release 16.1R2 and preceding, Python scripts must be owned by the root user.

4. Commit the configuration.

```
[edit]
user@host# commit
```

Preventing Commit Scripts from Executing During Commit Operations

You can prevent a commit script from executing during a commit operation by removing the script's filename from the `[edit system scripts commit]` hierarchy in the configuration.

> **NOTE**: You can also use the `deactivate` statement to deactivate a script instead of removing the script's filename from the configuration. Deactivated scripts may be reactivated later.

To prevent a commit script from executing during a commit operation:

1. Delete the commit script filename at the `[edit system scripts commit]` hierarchy level in the configuration.

```
[edit system scripts commit]
user@host# delete file filename
```

2. Commit your changes.

```
[edit]
user@host# commit
```

3. (Optional) Remove the commit script from the commit script directory on the device.

Although removing the commit script from the commit script directory is not required, it is always a good policy to delete unused files from the system.

Deactivating Commit Scripts

Deactivating a commit script results in its being marked as `inactive` in the configuration and ignored during a commit operation. You can reactivate the script by using the activate statement.

To deactivate a commit script so that it does not execute during the commit operation:

1. Issue the `deactivate` command.

```
[edit]
user@host# deactivate system scripts commit file filename
```

2. Commit your changes.

```
[edit]
user@host# commit
```

3. Verify that the commit script is deactivated.

```
[edit]
user@host# show system scripts commit
inactive: file ospf-neighbors.slax
```

Activating Commit Scripts

Deactivating a commit script results in its being marked as inactive in the configuration and is therefore not executed during the commit operation.

To activate an inactive commit script:

1. Issue the `activate` command.

```
[edit]
user@host# activate system scripts commit file filename
```

2. Commit your changes.

```
[edit]
user@host# commit
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
16.1R3	Starting in Junos OS Release 16.1R3, Python scripts can be owned by either root or a user in the Junos OS super-user login class.

Control the Execution of Commit Scripts in the QFabric System

IN THIS SECTION

This document describes the tasks that affect the way commit scripts are executed. In the QFabric system, commit scripts are stored in the in the **/pbdata/mgd_shared/*partition-ip*/var/db/scripts/commit** directory that is shared among Director devices in a Director group.

To determine which commit scripts are currently enabled on the QFabric system, use the `show` command to display the files included at the `[edit system scripts commit]` hierarchy level. To ensure that the enabled files are on the device, list the contents of the **/pbdata/mgd_shared/*partition-ip*/var/db/scripts/commit** directory using the `file list` operational mode command.

See the following tasks:

Enabling Commit Scripts to Execute

The commit operation requires that all scripts be included in configuration at the `[edit system scripts commit file]` hierarchy level for all QFabric Director devices.

If you need to temporarily remove a script from a commit operation but do not want to remove it from the configuration permanently, you may configure the `optional` statement at the `[edit system scripts commit file filename]` hierarchy level to enable the commit operation to succeed even if a script is missing from 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 has not been executed as expected.

The filename of a commit script written in SLAX or Python must include the **.slax** or **.py** file extension, respectively, for the script to be executed.

To enable a commit script to execute during a commit operation:

1. Ensure that the commit script is located in the correct directory: **/pbdata/mgd_shared/*partition-ip*/var/db/scripts/commit** directory on the Director device.
2. Configure the commit script.

```
[edit system scripts commit]
user@switch# set file filename <optional>
```

3. Commit the configuration.

```
[edit system scripts commit]
user@switch# top
[edit]
user@switch# commit
```

Removing Commit Scripts from the Configuration

You can prevent a commit script from executing during a commit operation by removing the script's filename from the `[edit system scripts commit]` hierarchy in the configuration.

NOTE: You can also deactivate a script using the `deactivate` statement instead of removing it from the configuration. Deactivated scripts may be reactivated later.

To prevent a commit script from executing during a commit operation:

1. Delete the commit script filename from the `[edit system scripts commit]` hierarchy level in the configuration.

```
[edit system scripts commit]
user@switch# delete file filename
```

2. Commit the configuration.

```
[edit system scripts commit]
user@switch# top
[edit]
user@switch# commit
```

3. (Optional) Remove the commit script from the **/pbdata/mgd_shared/** directory on the Director device.

BEST PRACTICE: Although removing the commit script is not necessary, we recommend deleting unused files from the system.

Deactivating Commit Scripts

Deactivating a commit script results in its being marked as inactive in the configuration. The script is not executed during the commit operation, but you can reactivate the script by using the `activate` statement.

To deactivate the commit script:

1. Deactivate the script.

```
[edit]
user@switch# deactivate system scripts commit file filename
```

2. Commit your changes.

```
[edit]
user@switch# commit
```

3. Verify that the commit script is deactivated.

```
[edit]
user@switch# show system scripts commit
inactive: file config-check.slax
```

Activating Inactive Commit Scripts

Deactivating a commit script results in its being marked as inactive in the configuration and is therefore not executed during the commit operation.

To activate an inactive commit script:

1. Activate the script.

```
[edit]
user@switch#  activate system scripts commit file filename
```

2. Commit your changes.

```
[edit]
user@switch# commit
```

RELATED DOCUMENTATION

Understanding Automation Script Support on the QFabric System Director Devices | 7

Configure 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.

   Starting in Junos OS Release 18.2R2 and 18.3R1, Junos OS supports only the SHA-256 hash function for configuring script checksum hashes. Earlier releases support the *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 and the `checksum` statement for one or more hash values.

```
[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, Junos OS recalculates the checksum value using the configured hash algorithm 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.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
18.3R1	Starting in Junos OS Release 18.2R2 and 18.3R1, Junos OS supports only the SHA-256 hash function for configuring script checksum hashes.

RELATED DOCUMENTATION

Configuring Checksum Hashes for an Event Script | 1101
Configure Checksum Hashes for an Op Script | 849
Configuring Checksum Hashes for an SNMP Script | 1130

How to Process Large Configurations Against Commit Scripts

In the standard commit model, when you perform a commit operation and commit scripts are in use, the management process (*mgd*) exports the post-inheritance candidate configuration in XML format and passes it as input to the commit script. The script driver then processes this configuration file against the configured commit scripts and returns any generated actions to the management process.

If the configuration is large, the script driver might have trouble reading the configuration into memory during the commit operation. When this occurs, you can configure the `direct-access` statement at the `[edit system scripts commit]` hierarchy level to enable the script driver to retrieve the *candidate configuration* directly from the configuration database. We recommend configuring the `direct-access` statement only if the configuration is large, because directly accessing the configuration data is more processor-intensive compared to the standard commit model and can affect system performance.

NOTE: Junos OS supports configuring the `direct-access` statement only when SLAX and XSLT commit scripts are configured. It does not support configuring the `direct-access` statement when Python commit scripts are also configured.

To enable 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 (Commit Scripts);
```

Example: Retrieve the Pre-Inheritance Candidate Configuration in a Commit Script

IN THIS SECTION

This example shows how to construct a commit script to retrieve the pre-inheritance candidate configuration for either a normal or private configuration session.

Requirements

- Routing, switching, or security device running Junos OS Release 12.2 or later.

Overview

In Junos OS, when a *candidate configuration* is committed, it is inspected by each active commit script. The normal input for a commit script is the post-inheritance candidate configuration, in which all configuration group values have been inherited by their targets and the inactive portions of the configuration have been removed.

At times, a commit script requires access to the pre-inheritance candidate configuration rather than the post-inheritance configuration it receives by default. Since normal configuration sessions use the regular candidate database, and private configuration sessions use a dedicated, private candidate database, the candidate database location depends on the session type.

Within a commit script, invoking the `<get-configuration>` remote procedure call (RPC) with the `database="candidate"` attribute retrieves the normal pre-inheritance candidate configuration. The `<get-configuration>` RPC also has a `database-path` attribute, which is used to specify the location of the pre-inheritance configuration database for either a normal or private configuration session. This attribute is an alternative to the `database` attribute and indicates which database file to load. Commit scripts can invoke the `<get-configuration>` RPC with the `database-path` attribute to retrieve the pre-inheritance candidate configuration specific to that session.

The global variable, `$junos-context` contains the `commit-context/database-path` element, which stores the location of the session's pre-inheritance candidate configuration. In a normal configuration session, the `<database-path>` element contains the location of the normal candidate database:

```
<junos-context>
    <commit-context>
        <database-path>/var/run/db/juniper.db</database-path>
    </commit-context>
</junos-context>
```

In a private configuration session, the `<database-path>` element contains the location of the session-specific, private candidate database. For example:

```
<junos-context>
    <commit-context>
        <commit-private/>
        <database-path>/var/run/db/private/juniper-1396.db</database-path>
    </commit-context>
</junos-context>
```

To construct a commit script that retrieves the pre-inheritance candidate configuration specific to that session, include the `<get-configuration>` RPC in the commit script, and set the `<database-path>` attribute to `$junos-context/commit-context/database-path`. For normal configuration sessions, the commit-script retrieves the normal pre-inheritance candidate configuration, and for private configuration sessions, the commit-script retrieves the private, pre-inheritance candidate configuration.

NOTE: If a commit script includes both the `database` and the `database-path` attributes in the `<get-configuration>` tag, the `database` attribute takes precedence.

Configuration

IN THIS SECTION

Configuring the Commit Script

Step-by-Step Procedure

To construct a commit script that retrieves the pre-inheritance candidate configuration specific to that session:

1. In a text editor, add the commit script boilerplate to a file.

```
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 {

}
```

2. Create a variable that stores the `<get-configuration>` RPC with the `database-path` attribute set to `$junos-context/commit-context/database-path`.

```
var $rpc = <get-configuration database-path=$junos-context/commit-context/database-path>;
```

3. Add a statement that invokes the `<get-configuration>` RPC and stores the resulting configuration in a variable.

```
var $config = jcs:invoke( $rpc );
```

4. Refer to the desired hierarchy levels and statements in the pre-inheritance candidate configuration using normal XPath constructs, for example:

```
var $hostname = $config/system/host-name;
```

5. Include any statements required to enforce your custom configuration rules during the commit process.

6. Copy the script to the **/var/run/scripts/commit** directory on the device.

7. In configuration mode, configure the `file` statement to enable the commit script.

```
[edit system scripts commit]
user@R1# set file script-name.slax
```

8. Issue the `commit` command to commit the configuration.

```
[edit]
user@R1# commit
```

The commit script is executed during the commit operation.

Results

```
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 $rpc = <get-configuration database-path=$junos-context/commit-context/database-path>;
    var $config = jcs:invoke( $rpc );
    ...
   <!-- commit script rules -->
}
```

RELATED DOCUMENTATION

Global Parameters and Variables in Junos OS Automation Scripts | 327

CHAPTER 23

Generate a Custom Warning, Error, or System Log Message Using Commit Scripts

IN THIS CHAPTER

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 Using Commit Scripts" on page 598.

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.

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 System Log Explorer 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.

RELATED DOCUMENTATION

Example: Generate a Custom Error Message | 586
Example: Generate a Custom System Log Message | 592
Example: Generate a Custom Warning Message | 580
Generate a Custom Warning, Error, or System Log Message in Commit Scripts | 572
SLAX and XSLT Commit Script Tag Elements to Use When Generating Messages | 577

Generate a Custom Warning, Error, or System Log Message in Commit Scripts

Junos OS commit scripts can generate custom warning, error, or system log messages during a commit operation to alert you when the configuration does not comply with custom configuration rules.

Generating an error also causes the commit operation to fail. To generate a custom warning, error, or system log message in a commit script:

1. Include the appropriate commit script boilerplate from "Required Boilerplate for Commit Scripts" on page 542. 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 1.2;
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
    */
}
```

   Python Boilerplate

```
from junos import Junos_Configuration
import jcs
```

```
if __name__ == '__main__':
    # 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 or Python equivalents. 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 `host-name` statement is not included at the `[edit system]` hierarchy level:

     ```
     <xsl:if test="not(system/host-name)">
     ```

     In SLAX, the `if` construct looks like this:

     ```
     if (not(system/host-name))
     ```

     Similarly in Python:

     ```
      if not(Junos_Configuration.xpath("./system/host-name")):
     ```

3. Include the appropriate constructs to generate a warning, error, or system log message.

In SLAX and XSLT scripts, include `<xnm:warning>`, `<xnm:error>`, or `<syslog>` elements 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.

In Python scripts, include `jcs.emit_warning()`, `jcs.emit_error()`, or `jcs.syslog()` functions, and include the message string in the argument list.

For example, when an XSLT commit script generates a warning using the following `<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>
```

it emits the following output during the `commit` operation:

```
[edit chassis]
    warning: IP source-route processing is not enabled.
commit complete
```

Similarly, when an XSLT commit script generates an error using the following `<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>
```

it emits the following output during the `commit` operation:

```
[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: In SLAX and XSLT scripts, 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 Using Commit Scripts" on page 598.)

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 "SLAX and XSLT Commit Script Tag Elements to Use When Generating Messages" on page 577.

For SLAX versions of these constructs, see "Example: Generate a Custom Warning Message" on page 580, "Example: Generate a Custom Error Message" on page 586, and "Example: Generate a Custom System Log Message" on page 592.

4. Save the script with a meaningful name.

5. Copy the script to either the **/var/db/scripts/commit** directory on the hard disk or the **/config/scripts/commit** directory on the flash drive.

   For information about setting the storage location for commit scripts, see "Store and Enable Junos Automation Scripts" on page 448 and "Store Scripts in Flash Memory" on page 451.

6. Enable the script by including the `file` *`filename`* statement at the `[edit system scripts commit]` hierarchy level.

```
[edit system scripts commit]
user@host# set file filename
```

7. If the script is written in Python, enable the execution of unsigned Python scripts.

```
[edit]
user@host# set system scripts language (python | python3)
```

8. Commit the configuration.

```
[edit]
user@host# commit and-quit
```

> **NOTE**: If the device has dual Routing Engines, and you want the script to take effect on both of them, you can issue the `commit synchronize scripts` command to synchronize the configuration and copy the scripts to the other Routing Engine as part of the commit operation.

RELATED DOCUMENTATION

SLAX and XSLT Commit Script Tag Elements to Use When Generating Messages | 577
Example: Generate a Custom Error Message | 586
Example: Generate a Custom System Log Message | 592
Example: Generate a Custom Warning Message | 580

SLAX and XSLT Commit Script Tag Elements to Use When Generating Messages

Junos OS commit scripts can generate custom warning, error, or system log messages during a commit operation to alert you when the configuration does not comply with custom configuration rules. Table 44 on page 577 summarizes the tag elements that you can include in a custom warning, error, or system log message in SLAX and XSLT commit scripts.

Table 44: 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.

Table 44: Tags and Attributes for Creating Custom Warning, Error, and System Log Messages *(Continued)*

Data Item, XML Element, or Attribute	Required or Supported	Description
`<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.
`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.

Table 44: Tags and Attributes for Creating Custom Warning, Error, and System Log Messages *(Continued)*

Data Item, XML Element, or Attribute	Required or Supported	Description
`<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.
`<token>`	Supported in warning and error messages only	Names the element in the request that caused the warning or error message.

Table 44: Tags and Attributes for Creating Custom Warning, Error, and System Log Messages *(Continued)*

Data Item, XML Element, or Attribute	Required or Supported	Description
`<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`.

For examples that use the tags to generate warnings, errors, and system log messages, see:

- "Example: Generate a Custom Warning Message" on page 580
- "Example: Generate a Custom Error Message" on page 586
- "Example: Generate a Custom System Log Message" on page 592

Example: Generate a Custom Warning Message

IN THIS SECTION

Junos OS commit scripts can generate custom warning messages during a commit operation to alert you when the configuration does not comply with custom configuration rules. The commit process is not affected by warnings. This example creates a commit script that generates a custom warning message when a specific statement is not included in the device configuration.

Requirements

Junos OS Release 16.1R3 or later release when using a Python script.

Overview and Commit Script

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.

The script is shown in XSLT, SLAX, and Python.

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 1.2;
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.";
        }
    }
}
```

Python Syntax

```
from junos import Junos_Configuration
import jcs

def main():
    root = Junos_Configuration
    if not(root.xpath("./chassis/source-route")):
        jcs.emit_warning("IP source-route processing is not enabled.")

if __name__ == '__main__':
    main()
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

Download, enable, and test the 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:

1. Copy the script into a text file, name the file **source-route.xsl**, **source-route.slax**, or **source-route.py** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

   > **NOTE**: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

2. In configuration mode, configure the `file` statement and the script filename at the `[edit system scripts commit]` hierarchy level.

```
[edit]
user@host# set system scripts commit file source-route.xsl
```

3. If the script is written in Python, enable the execution of unsigned Python scripts.

```
[edit]
user@host# set system scripts language python
```

   > **NOTE**: Configure the `language python3` statement to use Python 3 to execute Python scripts, or configure the `language python` statement to use Python 2.7 to execute Python scripts. For more information, see *language*.

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
```

Verification

IN THIS SECTION

Verifying Script Execution

Purpose

Verify the warning message generated by the commit script.

Action

Execute the `commit check` or `commit` command and review the output. The commit script generates a warning message when the `source-route` statement is not included at the `[edit chassis]` hierarchy level of the configuration. The warning does not affect the commit process.

```
[edit]
user@host# commit check
[edit chassis]
    warning: IP source-route processing is not enabled.
configuration check succeeds
```

```
[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
```

RELATED DOCUMENTATION

Example: Generate a Custom Error Message

IN THIS SECTION

Junos OS commit scripts can generate custom error messages during a commit operation to alert you when the configuration violates custom configuration rules. Emitting an error message causes the commit to fail. This example creates a commit script that generates a custom error message when a specific statement is not included in the device configuration, thereby halting the commit operation.

Requirements

Junos OS Release 16.1R3 or later release when using a Python script.

Overview and Commit Script

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:

The script is shown in XSLT, SLAX, and Python.

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 1.2;
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.";
            }
        }
    }
}
```

Python Syntax

```
from junos import Junos_Configuration
import jcs

def main():
    root = Junos_Configuration
    for element in root.xpath(
        "./interfaces/interface[starts-with(name,'t1-')]"):
        # Missing description
        if element.find('description') is None:
            # Emit error message to console
            jcs.emit_error("Missing a description for this T1 interface: "
                + element.find('name').text)

if __name__ == '__main__':
    main()
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

Download, enable, and test the 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:

1. Copy the script into a text file, name the file **description.xsl**, **description.slax**, or **description.py** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

NOTE: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

2. In configuration mode, configure the `file` statement and the script filename at the `[edit system scripts commit]` hierarchy level.

```
[edit]
user@host# set system scripts commit file description.xsl
```

3. If the script is written in Python, enable the execution of unsigned Python scripts.

```
[edit]
user@host# set system scripts language python
```

NOTE: Configure the `language python3` statement to use Python 3 to execute Python scripts, or configure the `language python` statement to use Python 2.7 to execute Python scripts. For more information, see *language*.

4. If the configuration for every T1 interface includes the `description` statement, delete the description for an existing T1 interface for testing purposes.

```
[edit]
user@host# delete interfaces t1-0/0/1 description
```

5. Issue the `commit` command to commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying Script Execution

Purpose

Verify the error message generated by the commit script.

Action

Review the output of the `commit` command. The commit script generates an error message for each T1 interface that does not include a `description` statement. Any error causes the commit process to fail.

```
[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
```

RELATED DOCUMENTATION

- Example: Generate a Custom System Log Message | **592**
- Example: Generate a Custom Warning Message | **580**
- Generate a Custom Warning, Error, or System Log Message in Commit Scripts | **572**

Example: Generate a Custom System Log Message

IN THIS SECTION

Junos OS commit scripts can generate custom system log messages during a commit operation to alert you when the configuration does not comply with custom configuration rules. The commit process is not affected by generating system log messages. This example creates a commit script that generates a custom system log message when a specific statement is not included in the device configuration.

Requirements

Junos OS Release 16.1R3 or later when using a Python script.

Overview and Commit Script

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.

The script is shown in XSLT, SLAX, and Python.

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) or (authorization != 'read-write')">
        <xsl:variable name="community">
          <xsl:call-template name="jcs:edit-path"/>
        </xsl:variable>
        <xsl:variable name="message" select="concat('SNMP community does not have read-write
access: ', $community)"/>
        <syslog>
          <message>
            <xsl:value-of select="$message"/>
          </message>
        </syslog>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

SLAX Syntax

```
version 1.2;
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) or (authorization != "read-write")) {
            var $community = call jcs:edit-path();
            var $message = "SNMP community does not have read-write access: " _ $community;
            <syslog> {
                <message> $message;
            }
        }
    }
}
```

Python Syntax

```
from junos import Junos_Configuration
import jcs

def main():
    root = Junos_Configuration
    for element in root.xpath("./snmp/community"):
        if element.find("authorization") is None or \
           element.find("authorization").text != 'read-write':
            jcs.syslog("172", "SNMP community does not have read-write access: "
                + element.find('name').text)

if __name__ == '__main__':
    main()
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

Download, enable, and test the script. 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. For 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:

1. Copy the script into a text file, name the file **read-write.xsl**, **read-write.slax**, or **read-write.py** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

   NOTE: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

2. In configuration mode, configure the `file` statement and the script filename at the `[edit system scripts commit]` hierarchy level.

```
[edit]
user@host# set system scripts commit file read-write.xsl
```

3. If the script is written in Python, enable the execution of unsigned Python scripts.

```
[edit]
user@host# set system scripts language python
```

   NOTE: Configure the `language python3` statement to use Python 3 to execute Python scripts, or configure the `language python` statement to use Python 2.7 to execute Python scripts. For more information, see *language*.

4. (Optional) To test the condition, if the `read-write` statement is included at the `[edit snmp community` *`community-name`* `authorization]` hierarchy level for every community, temporarily delete the authorization for an existing SNMP community.

```
[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 System Log Explorer.

6. Issue the `commit` command to commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying Script Execution

Purpose

Verify the system log message generated by the commit script.

NOTE: System log messages are generated during a commit operation for Python, SLAX, and XSLT scripts, but they are only generated during a commit check operation for Python scripts. This means you cannot use the `commit check | display xml` or `commit check |`

`display detail` configuration mode commands to verify the output of system log messages for SLAX and XSLT scripts.

Action

When the commit operation completes, inspect the system log file. The default directory for log files is **/var/log/**. View the log file by issuing the `show log` *`filename`* operational mode command. For example, if messages are logged to the **messages** file, issue the following command:

```
user@host> show log messages | match cscript
```

System log entries generated by commit scripts have the following format:

```
timestamp host-name cscript: message
```

Since the `read-write` statement was not included at the `[edit snmp community` *`community-name`* `authorization]` hierarchy level, the commit script should generate the “SNMP community does not have read-write access” message in the system log file.

```
Jun 3 14:34:37 host-name cscript: SNMP community does not have read-write access: [edit snmp
community community-name]
```

RELATED DOCUMENTATION

Example: Generate a Custom Error Message | 586
Example: Generate a Custom Warning Message | 580
Generate a Custom Warning, Error, or System Log Message in Commit Scripts | 572

CHAPTER 24

Generate Persistent or Transient Configuration Changes Using Commit Scripts

IN THIS CHAPTER

Overview of Generating Persistent or Transient Configuration Changes Using Commit Scripts

IN THIS SECTION

Junos OS commit scripts enforce custom configuration rules. When a *candidate configuration* includes statements that you have decided must not be included in your configuration, or when the candidate configuration omits statements that you have decided are required, commit scripts can automatically change the configuration and thereby correct the problem.

Differences Between Persistent and Transient Changes

Configuration changes made by commit scripts can be one of the following types:

- **persistent**—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.

- **transient**—A *transient change* 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 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, 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, 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 OS 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.

Protected elements in the configuration hierarchy cannot be modified or deleted by either a persistent or a transient change. If a commit script attempts to modify or delete a protected statement or hierarchy, Junos OS issues a warning that the change cannot be made, and proceeds with the commit.

Persistent and transient changes have several important differences, as described in Table 45 on page 600.

Table 45: Differences Between Persistent and Transient Changes

Persistent Changes	Transient Changes
You can represent a persistent change in commit scripts by using the `content` parameter in conjunction with a `tag` parameter that is set to 'change' inside a call to the `jcs:emit-change` template in SLAX and XSLT scripts or a call to the `jcs.emit_change` method in Python scripts. SLAX and XSLT commit scripts can also represent a persistent change by using the `<change>` tag.	You can represent a transient change in commit scripts with the `content` parameter in conjunction with the a `tag` parameter that is set to 'transient-change' inside a call to the `jcs:emit-change` template in SLAX and XSLT scripts or a call to the `jcs.emit_change` method in Python scripts. SLAX and XSLT commit scripts can also represent a transient change by using the `<transient-change>` tag.
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.	In order to load transient changes, you must configure the `allow-transients` statement. You can configure the statement at the `[edit system scripts commit]` hierarchy level for all commit scripts. Additionally, on supported platforms and releases, you can configure the `allow-transients` statement at the `[edit system scripts commit file *filename*]` hierarchy level for individual commit scripts. 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. You cannot use a commit script to generate the `allow-transients` statement. Like persistent changes, transient changes must pass the standard Junos OS validity check.

Table 45: Differences Between Persistent and Transient Changes *(Continued)*

Persistent Changes	Transient Changes
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, 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, 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.
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.

Table 45: Differences Between Persistent and Transient Changes *(Continued)*

Persistent Changes	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 538.
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 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 CLI User Guide .

Tag Elements and Templates for Generating Changes

To generate persistent or transient changes in commit scripts, SLAX and XSLT scripts can use the `jcs:emit-change` template, and Python scripts can use the `jcs.emit_change` method. The `jcs:emit-change` template and `jcs.emit_change` method implicitly include `<change>` and `<transient-change>` XML elements. SLAX and XSLT scripts can also generate changes by including the `<change>` and `<transient-change>` elements directly in the commit script. Using the `jcs:emit-change` template in SLAX and XSLT scripts allows you to set the hierarchical context of the change once rather than multiple times. In Python scripts, the `jcs.emit_change` method requires that the configuration data for the requested change include the full configuration path representing all levels of the configuration hierarchy formatted as an XML string.

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 Developer Guide* .

RELATED DOCUMENTATION

Generate a Persistent or Transient Configuration Change in SLAX and XSLT Commit Scripts

Junos OS commit scripts enforce custom configuration rules and can automatically change the configuration when it does not comply with your custom configuration rules. To generate a *persistent change* or *transient change* in Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative syntaX (SLAX) commit scripts:

1. At the start of the script, include the XSLT or SLAX commit script boilerplate from "Required Boilerplate for Commit Scripts" on page 542.

   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 1.2;
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. 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 present, 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 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 equivalent `for-each` and `if` constructs are:

   ```
   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 34. For information about writing scripts in SLAX instead of XSLT, see "SLAX Overview" on page 83.

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.

```
<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.

```
<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 must be `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.

```
<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 disk or the **/config/scripts/commit** directory on the flash drive. For information about setting the storage location for commit scripts, see "Store Scripts in Flash Memory" on page 451.

   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 both Routing Engines. The commit synchronize command does not copy scripts between Routing Engines.

6. Enable the script by including the `file filename` statement at the `[edit system scripts commit]` hierarchy level.

```
[edit system scripts commit]
user@host# set file filename
```

7. If the script generates transient changes, configure the `allow-transients` statement.

   Configure the statement at the `[edit system scripts commit]` hierarchy level to enable all commit scripts to make transient changes.

```
[edit system scripts commit]
user@host# set allow-transients
```

   Alternatively, on supported devices and releases, configure the statement at the `[edit system scripts commit file filename]` hierarchy level to enable only the individual script to make transient changes.

```
[edit system scripts commit]
user@host# set file filename allow-transients
```

8. Commit the configuration.

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

If all the commit scripts run without errors, any persistent changes are loaded into the *candidate configuration*. Any transient changes are loaded into the checkout configuration, but not to 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. However, persistent changes are loaded into the candidate configuration, and transient changes are loaded into the checkout configuration.

RELATED DOCUMENTATION

SLAX and XSLT Commit Script Tag Elements to Use When Generating Persistent and Transient Configuration Changes | **614**

Remove a Persistent or Transient Configuration Change Using Commit Scripts | **615**

Example: Generate Persistent and Transient Configuration Changes Using Commit Scripts | **617**

Generate a Persistent or Transient Configuration Change in Python Commit Scripts | **610**

emit-change Template (SLAX and XSLT) and emit_change (Python) | **434**

Generate a Persistent or Transient Configuration Change in Python Commit Scripts

Junos OS commit scripts enforce custom configuration rules and can automatically change the configuration when it does not comply with your custom configuration rules. To generate a *persistent change* or *transient change* using Python commit scripts:

1. At the start of the script, include the Python boilerplate from "Required Boilerplate for Commit Scripts" on page 542, which is reproduced here for convenience:

```
from junos import Junos_Configuration
import jcs
```

```
if __name__ == '__main__':
    # insert your code here
```

2. Include one or more programming instructions that test for your custom configuration rules.

   For example, the following code selects each SONET/SDH interface that does not have the MPLS protocol family enabled:

```
    # Get configuration root object
    root = Junos_Configuration

    for element in root.xpath("./interfaces/ \
        interface[starts-with(name,'so-')]"):
        if element.find('unit/family/mpls') is None:
```

3. Create an XML string that instructs Junos OS how to modify the configuration.

   This example enables the MPLS protocol family for the selected interfaces.

```
            if_name = element.find('name').text
            unit_name = element.find('unit/name').text
            change_xml = """
            <interfaces>
               <interface>
                  <name>{0}</name>
                  <unit>
                     <name>{1}</name>
                     <family>
                        <mpls>
                        </mpls>
                     </family>
                  </unit>
               </interface>
            </interfaces>
            """.format(if_name, unit_name).strip()
```

4. To generate the persistent or transient change, call the `jcs.emit_change` method, and specify the type of change, either 'change' or 'transient-change', in the argument list.

```
            jcs.emit_change(change_xml, "change", "xml")
```

5. Include any additional required or optional code. This example generates a warning message that is displayed on the CLI when the commit script updates a SONET/SDH interface.

```
jcs.emit_warning("Adding 'family mpls' to SONET interface: " + if_name)
```

6. Save the script with a meaningful name.
7. Copy the script to either the **/var/db/scripts/commit** directory on the device hard disk or the **/config/scripts/commit** directory on the flash drive.

> **NOTE**: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

8. Enable the script by configuring the `file` *`filename`* statement at the `[edit system scripts commit]` hierarchy level.

```
[edit]
user@host# set system scripts commit file filename
```

9. If the script generates transient changes, configure the `allow-transients` statement.

   Configure the statement at the `[edit system scripts commit]` hierarchy level to enable all commit scripts to make transient changes.

```
[edit system scripts]
user@host# set commit allow-transients
```

   Alternatively, on supported devices and releases, configure the statement at the `[edit system scripts commit file` *`filename`*`]` hierarchy level to enable only the individual script to make transient changes.

```
[edit system scripts]
user@host# set commit file filename allow-transients
```

10. Enable the execution of unsigned Python scripts on the device.

```
[edit system scripts]
user@host# set language (python | python3)
```

11. Commit the configuration.

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

The resulting 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 as a persistent change, and emits a warning message stating that the configuration has been changed.

```
from junos import Junos_Configuration
import jcs

def main():
    # Get configuration root object
    root = Junos_Configuration

    for element in root.xpath("./interfaces/ \
        interface[starts-with(name,'so-')]"):
        if element.find('unit/family/mpls') is None:
            if_name = element.find('name').text
            unit_name = element.find('unit/name').text
            change_xml = """
            <interfaces>
               <interface>
                  <name>{0}</name>
                  <unit>
                     <name>{1}</name>
                     <family>
                        <mpls>
                        </mpls>
                     </family>
                  </unit>
               </interface>
            </interfaces>
            """.format(if_name, unit_name).strip()
            jcs.emit_change(change_xml, "change", "xml")
            jcs.emit_warning("Adding 'family mpls' to SONET interface: " + if_name)
```

```
if __name__ == '__main__':
    main()
```

If all enabled commit scripts run without errors, any persistent changes are loaded into the *candidate configuration*. Any transient changes are loaded into the checkout configuration, but not to 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. However, persistent changes are loaded into the candidate configuration, and transient changes are loaded into the checkout configuration.

RELATED DOCUMENTATION

Generate a Persistent or Transient Configuration Change in SLAX and XSLT Commit Scripts | 604
emit-change Template (SLAX and XSLT) and emit_change (Python) | 434

SLAX and XSLT Commit Script Tag Elements to Use When Generating Persistent and Transient Configuration Changes

Junos OS commit scripts enforce custom configuration rules and can automatically change the configuration when it does not comply with your custom configuration rules. Table 46 on page 615 summarizes the tag elements that you can include in SLAX and XSLT commit scripts to generate persistent and transient changes. To see how data values are supplied within a script, see "Example: Generate Persistent and Transient Configuration Changes Using Commit Scripts" on page 617.

Table 46: Tags and Attributes for Creating Configuration Changes in SLAX and XSLT Commit Scripts

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 checkout 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`.
`<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.

Remove a Persistent or Transient Configuration Change Using Commit Scripts

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 disables 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:

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. Commit the configuration.

   ```
   [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 disk or **/config/**

scripts/commit on the flash drive. 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
```

RELATED DOCUMENTATION

Overview of Generating Persistent or Transient Configuration Changes Using Commit Scripts | 598
Store Scripts in Flash Memory | 451

Example: Generate Persistent and Transient Configuration Changes Using Commit Scripts

IN THIS SECTION

Example: Generating a Persistent Change

IN THIS SECTION

Junos OS commit scripts enable users to customize the validation process of their configurations in accordance with their own practices and policies and enforce custom configuration rules during the commit process. This example demonstrates a commit script that 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. If you do not explicitly configure the MPLS protocol family on an interface, the interface is not enabled for MPLS applications.

Requirements

This example uses the following hardware and software components:

- Device running Junos OS with one or more SONET/SDH interfaces.
- Junos OS Release 16.1R3 or later release when using a Python script.

Overview and Commit Script

The commit script in this example finds all SONET/SDH interfaces that have a logical interface configured but that do not have the `family mpls` statement configured. For these interfaces, the script adds the `family mpls` statement to the interface configuration as a persistent change at the `[edit interfaces` *`interface-name`* `unit` *`logical-unit-number`*`]` hierarchy level. The script is shown in SLAX, XSLT, and Python.

The SLAX and XSLT versions of the commit script generate the persistent change by using the `jcs:emit-change` template, which is a helper template contained in the **junos.xsl** import file. The `tag` parameter of the `jcs:emit-change` template is omitted, which directs the script to emit the change as a persistent change. The `content` parameter of the `jcs:emit-change` template includes the configuration statements to add as a persistent change. The `message` parameter of the `jcs:emit-change` template includes the warning message to be displayed in the CLI, notifying you that the configuration has been changed.

The Python version of the commit script generates the persistent change by using the `jcs.emit_change()` function, which is imported from the `jcs` module. The Python script indicates that this is a persistent change by passing in the positional argument 'change'.

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>;
                    }
                }
            }
        }
    }
}
```

Python Syntax

```
from junos import Junos_Configuration
import jcs

def main():
    # Get configuration root object
    root = Junos_Configuration

    for element in root.xpath("./interfaces/ \
        interface[starts-with(name,'so-')]/unit"):
        if element.find('family/mpls') is None:
            if_name = element.find('../name').text
            unit_name = element.find('name').text
            change_xml = """
            <interfaces>
               <interface>
                  <name>{0}</name>
                  <unit>
                     <name>{1}</name>
                     <family>
                        <mpls>
                        </mpls>
                     </family>
                  </unit>
               </interface>
            </interfaces>
            """.format(if_name, unit_name).strip()
            jcs.emit_change(change_xml, "change", "xml")
            jcs.emit_warning("Adding 'family mpls' to SONET/SDH interface: " + if_name)
```

```
if __name__ == '__main__':
    main()
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script.

1. Copy the script into a text file, name the file **mpls.xsl**, **mpls.slax**, or **mpls.py** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

   NOTE: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

2. In configuration mode, configure the `file` statement and the script filename at the `[edit system scripts commit]` hierarchy level.

   ```
   [edit]
   user@host# set system scripts commit file mpls.xsl
   ```

3. If the script is written in Python, enable the execution of unsigned Python scripts.

   ```
   [edit]
   user@host# set system scripts language python
   ```

NOTE: Configure the `language python3` statement to use Python 3 to execute Python scripts, or configure the `language python` statement to use Python 2.7 to execute Python scripts. For more information, see *language*.

4. To test that the commit script generates the persistent change correctly, make sure that the configuration contains the condition that elicits the change. To test this script, ensure that the `family mpls` statement is not included at the `[edit interfaces so-`*`fpc/pic/port`*` unit `*`logical-unit-number`*`]` hierarchy level for at least one SONET/SDH interface.

5. Issue the `commit check` command to preview a trace of commit script processing to verify that the script will add the persistent change to the candidate configuration. The `commit check` command verifies the syntax of the configuration prior to a commit, but it does not commit the changes.

   The commit script in this example produces a message for each change it makes. Use the `commit check` command to preview these messages to determine whether the script will update the configuration with the `family mpls` statement for the appropriate interfaces.

   Issue the `commit check | display xml` command to display the XML-formatted version of the message. The sample output indicates that the script will add the `family mpls` statement to the so-2/3/4.0 interface configuration during the commit operation.

```
[edit]
user@host# commit check | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/11.2R1/junos">
    <commit-results>
        <routing-engine junos:style="normal">
            <name>re0</name>
            <xnm:warning xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
                <edit-path>
                    [edit interfaces interface so-2/3/4 unit 0]
                </edit-path>
                <message>
                    Adding 'family mpls' to SONET/SDH interface.
                </message>
            </xnm:warning>
            <commit-check-success/>
        </routing-engine>
    </commit-results>
</rpc-reply>
```

6. To display a detailed trace of commit script processing, issue the `commit check | display detail` command. In the sample output, there is one persistent change that will be loaded into the configuration during the commit operation.

```
[edit]
user@host# commit check | display detail
2011-06-17 14:17:35 PDT: reading commit script configuration
2011-06-17 14:17:35 PDT: testing commit script configuration
2011-06-17 14:17:35 PDT: opening commit script '/var/db/scripts/commit/mpls.xsl'
2011-06-17 14:17:35 PDT: reading commit script 'mpls.xsl'
2011-06-17 14:17:35 PDT: running commit script 'mpls.xsl'
2011-06-17 14:17:35 PDT: processing commit script 'mpls.xsl'
2011-06-17 14:17:35 PDT: no errors from mpls.xsl
2011-06-17 14:17:35 PDT: saving commit script changes for script mpls.xsl
2011-06-17 14:17:35 PDT: summary of script mpls.xsl: changes 1, transients 0, syslog 0
2011-06-17 14:17:35 PDT: start loading commit script changes
2011-06-17 14:17:35 PDT: loading commit script changes into real db
2011-06-17 14:17:35 PDT: finished commit script changes into real db
2011-06-17 14:17:35 PDT: no transient commit script changes
2011-06-17 14:17:35 PDT: finished loading commit script changes
2011-06-17 14:17:35 PDT: copying juniper.db to juniper.data+
2011-06-17 14:17:35 PDT: finished copying juniper.db to juniper.data+
...
configuration check succeeds
```

7. After verifying that the script produces the correct changes, issue the `commit` command to start the commit operation and execute the script.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the correct changes are integrated into the configuration.

Action

After executing the commit operation, view the configuration by issuing 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, the output is similar to the following:

```
[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

IN THIS SECTION

This example uses a commit script to set PPP encapsulation on all SONET/SDH interfaces with the IPv4 protocol family enabled. The changes are added as transient changes.

Requirements

This example uses the following hardware and software components:

- Device running Junos OS with one or more SONET/SDH interfaces.
- Junos OS Release 16.1R3 or later release when using a Python script.

Overview and Commit Script

The commit script in this example finds all SONET/SDH interfaces with the IPv4 protocol family enabled in the configuration and adds the `encapsulation ppp` statement to the interface configuration. The commit script generates a transient change, which adds the change to the checkout configuration but not the candidate configuration. The script is shown in SLAX, XSLT, and Python.

The SLAX and XSLT versions of the commit script generate the transient change by using the `jcs:emit-change` template, which is a helper template contained in the **junos.xsl** import file. The `tag` parameter of the `jcs:emit-change` template has the value `transient-change`, which directs the script to emit the change as a *transient change* rather than a *persistent change*. The `content` parameter of the `jcs:emit-change` template includes the configuration statements to be added as a transient change.

The Python version of the commit script generates the transient change by using the `jcs.emit_change()` function, which is imported from the `jcs` module. The Python script indicates that this is a transient change by passing in the positional argument 'transient-change'.

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";
            }
        }
    }
}
```

Python Syntax

```
from junos import Junos_Configuration
import jcs

def main():
    # Get configuration root object
    root = Junos_Configuration

    for element in root.xpath("./interfaces/ \
        interface[starts-with(name,'so-') and unit/family/inet]"):
        if_name = element.find('name').text
        change_xml = """
        <interfaces>
           <interface>
              <name>{0}</name>
              <encapsulation>ppp</encapsulation>
           </interface>
        </interfaces>
```

```
        """.format(if_name).strip()
        jcs.emit_change(change_xml, "transient-change", "xml")
        jcs.emit_warning("Adding 'ppp' encapsulation to SONET interface: " + if_name)
        jcs.emit_warning(change_xml)

if __name__ == '__main__':
    main()
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script.

1. Copy the script into a text file, name the file **encap-ppp.xsl**, **encap-ppp.slax**, or **encap-ppp.py** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

   NOTE: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

2. In configuration mode, configure the `file` statement and the script filename at the `[edit system scripts commit]` hierarchy level.

   ```
   [edit]
   user@host# set system scripts commit file encap-ppp.xsl
   ```

3. Configure the `allow-transients` statement to enable commit scripts to load transient changes into the checkout configuration.

```
[edit]
user@host# set system scripts commit allow-transients
```

4. If the script is written in Python, enable the execution of unsigned Python scripts.

```
[edit]
user@host# set system scripts language python3
```

> **NOTE**: Configure the `language python3` statement to use Python 3 to execute Python scripts, or configure the `language python` statement to use Python 2.7 to execute Python scripts. For more information, see *language*.

5. To test that the commit script generates the transient change correctly, make sure that the configuration contains the condition that elicits the change. Ensure that the `encapsulation ppp` statement is not included at the `[edit interfaces so-`*fpc/pic/port*`]` hierarchy level for at least one SONET/SDH interface.

6. Issue the `commit check` command to preview a trace of commit script processing to verify that the script will add the transient change to the checkout configuration. The `commit check` command verifies the syntax of the configuration prior to a commit, but it does not commit the changes.

   Issue the `commit check | display detail` command to display a detailed trace of commit script processing. In the sample output, there are two transient changes that are loaded into the checkout configuration.

```
[edit]
user@host# commit check | display detail
2011-06-15 12:07:30 PDT: reading commit script configuration
2011-06-15 12:07:30 PDT: testing commit script configuration
2011-06-15 12:07:30 PDT: opening commit script '/var/db/scripts/commit/encap-ppp.xsl'
2011-06-15 12:07:30 PDT: reading commit script 'encap-ppp.xsl'
2011-06-15 12:07:30 PDT: running commit script 'encap-ppp.xsl'
2011-06-15 12:07:30 PDT: processing commit script 'encap-ppp.xsl'
2011-06-15 12:07:30 PDT: no errors from encap-ppp.xsl
2011-06-15 12:07:30 PDT: saving commit script changes for script encap-ppp.xsl
2011-06-15 12:07:30 PDT: summary of script encap-ppp.xsl: changes 0, transients 2 (allowed),
```

```
syslog 0
2011-06-15 12:07:30 PDT: start loading commit script changes
2011-06-15 12:07:30 PDT: no commit script changes
2011-06-15 12:07:30 PDT: updating transient changes into transient tree
2011-06-15 12:07:30 PDT: finished loading commit script changes
2011-06-15 12:07:30 PDT: copying juniper.db to juniper.data+
2011-06-15 12:07:30 PDT: finished copying juniper.db to juniper.data+
2011-06-15 12:07:30 PDT: exporting juniper.conf
2011-06-15 12:07:30 PDT: merging transient changes
...
configuration check succeeds
```

7. After verifying that the script produces the correct changes, issue the `commit` command to start the commit operation and execute the script.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the correct changes are integrated into the checkout configuration. If there are one or more SONET/SDH interfaces with the IPv4 protocol family enabled, you should see the `encapsulation ppp` statement added as a *transient change* to the interface hierarchy.

Action

To view the configuration with transient changes, issue the `show interfaces | display commit-scripts` configuration mode command. The `show interfaces | display commit-scripts` command displays all the statements that are in the configuration, including statements that are generated by transient changes. If there are one or more SONET/SDH interfaces with the IPv4 protocol family enabled, the output is similar to the following:

```
[edit]
user@host# show interfaces | display commit-scripts
...
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 the script, because IPv4 protocol
                              # family is not configured on this interface.
    unit 0 {
        family mpls;
    }
}
```

Troubleshooting

IN THIS SECTION

Troubleshooting Commit Errors

Problem

The CLI generates an invalid transient change error, and the commit fails.

```
user@host# commit check
error: invalid transient change generated by commit script: encap-ppp.xsl
warning: 1 transient change was generated without [system scripts commit allow-transients]
error: 1 error reported by commit scripts
error: commit script failure
```

Solution

You must configure the `allow-transients` statement at the `[edit system scripts commit]` hierarchy level to enable commit scripts to load transient changes into the checkout configuration.

Configure the following statement to allow transient changes:

```
[edit]
user@host# set system scripts commit allow-transients
```

RELATED DOCUMENTATION

Generate a Persistent or Transient Configuration Change in SLAX and XSLT Commit Scripts | 604
Overview of Generating Persistent or Transient Configuration Changes Using Commit Scripts | 598
Remove a Persistent or Transient Configuration Change Using Commit Scripts | 615
emit-change Template (SLAX and XSLT) and emit_change (Python) | 434

CHAPTER 25

Create Custom Configuration Syntax with Commit Script Macros

IN THIS CHAPTER

Overview of Creating Custom Configuration Syntax with Commit Script 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 the Junos OS configuration and automatically correct configuration mistakes when they occur (as discussed in "Overview of Generating Persistent or Transient Configuration Changes Using Commit Scripts" on page 598). 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 Configure Logical Interfaces and IP Addresses" on page 674.

RELATED DOCUMENTATION

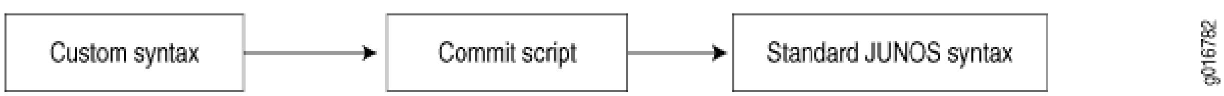
Create Custom Configuration Syntax with Commit Script Macros | 633

Create Custom Configuration Syntax with Commit Script Macros

IN THIS SECTION

Commit script macros enable you to create custom configuration syntax and expand it into standard Junos OS configuration statements. 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 7 on page 633. The standard Junos OS statements are added to the configuration to cause your intended operational changes.

Figure 7: Macro Input and Output



Macros use either persistent or *transient change* elements to expand your custom syntax into standard Junos OS configuration statements. If you use persistent changes, both the custom syntax and the standard Junos OS syntax appear in the candidate configuration. 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.

This section discusses the following topics:

Creating a Custom Syntax

Macros work by locating `apply-macro` statements 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, for example:

```
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 the corresponding persistent or transient change.

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, you can extract and manipulate the parameters contained in `apply-macro` statements by referring to the `<data>`, `<name>`, and `<value>` elements.

In the following example, the `select` attribute's *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"/>
```

The SLAX equivalent is:

```
var $color = ./data[name='color']/value;
```

The Python equivalent, which assumes that `element` has selected an `apply-macro` element, is:

```
color = element.find("data[name='color']/value").text
```

Expanding the Custom Syntax

In the corresponding commit script, you include one or more 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']) {
```

The Python equivalent, which spans multiple lines for readability, is:

```
for element in Junos_Configuration.xpath \
      ("./protocols/mpls/apply-macro[data/name='color']"):
```

When expanding macros, a particularly useful programming instruction in XSLT scripts 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 persistent or 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;
                    }
                }
            }
        }
    }
}
```

Similarly in Python:

```
    lsp_config =""
    for element2 in element.xpath("data[not(value)]/name"):
        lsp_config = lsp_config + """
        <label-switched-path>
           <name>{0}-lsp-{1}</name>
           <to>{1}</to>
           <admin-group>
              <include-any>{0}</include-any>
           </admin-group>
        </label-switched-path>
        """.format(color, element2.text)
```

```
    change_xml = """
    <protocols>
       <mpls>
          <admin-groups>
             <name>{0}</name>
             <group-value>{1}</group-value>
          </admin-groups>
          {2}
       </mpls>
    </protocols>
    """.format(color, group_value, lsp_config).strip()

    jcs.emit_change(change_xml, "transient-change", "xml")
```

NOTE: The example shown here is partial. For a full example, see "Example: Creating Custom Configuration Syntax with Commit Script Macros" on page 644.

After committing the configuration, the script runs, and the resulting full configuration looks like this:

```
[edit]
protocols {
    mpls {
        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;
        }
```

```
    }
}
```

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 633 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 633, 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: Control LDP Configuration" on page 716.

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.

RELATED DOCUMENTATION

Overview of Creating Custom Configuration Syntax with Commit Script Macros | 632

Create a Commit Script Macro to Read the Custom Syntax and Generate Related Configuration Statements | 641

Example: Creating Custom Configuration Syntax with Commit Script Macros | 644

Create a Commit Script Macro to Read the Custom Syntax and Generate Related Configuration Statements

Commit script macros enable you to expand custom configuration syntax into standard Junos OS 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 Junos OS statements.

To write such a script:

1. At the start of the script, include the appropriate commit script boilerplate from "Required Boilerplate for Commit Scripts" on page 542. 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 1.2;
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
    */
}
```

   Python Boilerplate

```
from junos import Junos_Configuration
import jcs
```

```
if __name__ == '__main__':
    # insert your code here
```

2. At the position indicated by the comment "*insert your code here*," include programming instructions that inspect the configuration for the `apply-macro` statement at a specified hierarchy level and change the configuration to include standard Junos OS syntax.

   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 Commit Script Macros" on page 644.

3. Save the script with a meaningful name.

4. Copy the script to either the **/var/db/scripts/commit** directory on the hard disk or the **/config/scripts/commit** directory on the flash drive.

   For information about setting the storage location for commit scripts, see "Store and Enable Junos Automation Scripts" on page 448 and "Store Scripts in Flash Memory" on page 451.

5. Enable the script by configuring the `file` *`filename`* statement at the `[edit system scripts commit]` hierarchy level.

```
[edit system scripts]
user@host# set commit file filename
```

6. If the script generates transient changes, configure the `allow-transients` statement.

   Configure the statement at the `[edit system scripts commit]` hierarchy level to enable all commit scripts to make transient changes.

```
[edit system scripts]
user@host# set commit allow-transients
```

   Alternatively, on supported devices and releases, configure the statement at the `[edit system scripts commit file` *`filename`*`]` hierarchy level to enable only the individual script to make transient changes.

```
[edit system scripts]
user@host# set commit file filename allow-transients
```

7. If the script is written in Python, enable the execution of unsigned Python scripts.

```
[edit system scripts]
user@host# set language (python | python3)
```

8. Commit the configuration.

```
[edit system scripts]
user@host# commit and-quit
```

If all the commit scripts run without errors, any persistent changes are loaded into the candidate configuration, and any transient changes are loaded into the checkout configuration, but not to 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.

RELATED DOCUMENTATION

Overview of Creating Custom Configuration Syntax with Commit Script Macros | 632
Create Custom Configuration Syntax with Commit Script Macros | 633
Example: Creating Custom Configuration Syntax with Commit Script Macros | 644

Example: Creating Custom Configuration Syntax with Commit Script Macros

IN THIS SECTION

A Junos OS configuration can contain `apply-macro` statements with custom configuration syntax. By itself, the `apply-macro` statement has no operational impact on the device. Commit script macros process the custom configuration syntax and expand it into standard Junos OS configuration statements, which are then added as a persistent or transient change. This example demonstrates how to use commit script macros to inspect `apply-macro` statements and generate Junos OS configuration statements.

Requirements

This example uses the following hardware and software components:

- Device running Junos OS.
- Junos OS Release 16.1R3 or later release when using a Python script.

Overview and Commit Script

Table 47 on page 645 shows a macro containing custom syntax and the corresponding expansion to standard Junos OS command-line interface (CLI) syntax.

Table 47: Sample Macro and Junos OS CLI Expansion

<table>
<tr><th>Custom Macro Syntax</th><th>Expanded Junos OS CLI Syntax</th></tr>
<tr><td><pre>
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;
        }
    }
}
</pre></td><td><pre>
protocols {
    mpls {
        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;
        }
    }
}
</pre></td></tr>
</table>

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 Table 47 on page 645 and a line-by-line explanation of the script:

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="$mpls/apply-macro[data/name = 'color']">
            <xsl:variable name="color" select="data[name = 'color']/value"/>
            <xsl:variable name="group-value" select="data[name = \
                       'group-value']/value"/>
            <transient-change>
                <protocols>
                    <mpls>
                        <admin-groups>
                            <name>
                                <xsl:value-of select="$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>
```

```
32                                              <xsl:value-of select="$color"/>
33                                          </include-any>
34                                      </admin-group>
35                                  </label-switched-path>
36                              </xsl:for-each>
37                          </mpls>
38                      </protocols>
39                  </transient-change>
40              </xsl:for-each>
41          </xsl:template>
42      </xsl:stylesheet>
```

Lines 1 through 8 (and Lines 43 and 44) are the boilerplate that you include in every XSLT 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 Table 47 on page 645 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 assigns the value of the `group-value` parameter, in this case `0`, to a variable called `group-value`.

```
12                  <xsl:variable name="group-value" select="data[name = \
                               'group-value']/value"/>
```

Lines 13 through 15 generate a transient change at the `[edit protocols mpls]` hierarchy level.

```
13                  <transient-change>
14                      <protocols>
15                          <mpls>
```

Lines 16 through 23 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.

```
16                              <admin-groups>
17                                  <name>
18                                      <xsl:value-of select="$color"/>
19                                  </name>
20                                  <group-value>
21                                      <xsl:value-of select="$group-value"/>
22                                  </group-value>
23                              </admin-groups>
```

The resulting configuration statements are as follows:

```
admin-groups {
    blue 0;
}
```

Line 24 selects the name of every parameter that does not 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 have a value assigned (blue and 0, respectively), so this line does not apply to them.

```
24                              <xsl:for-each select="data[not(value)]/name">
```

Line 25 adds the label-switched-path statement in the configuration.

```
25                                  <label-switched-path>
```

Lines 26 through 28 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 24 (represented by the " .").

```
26                                      <name>
27                                          <xsl:value-of select="concat($color, '-lsp-', .)"/>
28                                      </name>
```

Line 29 adds the `to` statement to the configuration and sets its value to the IP address currently selected by Line 24.

```
29                                      <to><xsl:value-of select="."/></to>
```

Lines 30 through 34 add the `admin-group include-any` statement to the configuration and set its value to the value of the `color` variable.

```
30                                      <admin-group>
31                                          <include-any>
32                                              <xsl:value-of select="$color"/>
33                                          </include-any>
34                                      </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 35 through 42 are closing tags.

```
35                                  </label-switched-path>
36                              </xsl:for-each>
37                          </mpls>
38                      </protocols>
39                  </transient-change>
40              </xsl:for-each>
41          </xsl:template>
42      </xsl:stylesheet>
```

SLAX Syntax

The equivalent SLAX script is:

```
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;
        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;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Python Syntax

The following Python syntax produces the identical configuration changes:

```
from junos import Junos_Configuration
import jcs

def main():
    # Get configuration root object
    root = Junos_Configuration

    for element in root.xpath("./protocols/mpls/ \
```

```
        apply-macro[data/name='color']"):
        color = element.find("data[name='color']/value").text
        group_value = element.find("data[name='group-value']/value").text

        lsp_config =""
        for element2 in element.xpath("data[not(value)]/name"):
            lsp_config = lsp_config + """
            <label-switched-path>
               <name>{0}-lsp-{1}</name>
               <to>{1}</to>
               <admin-group>
                  <include-any>{0}</include-any>
               </admin-group>
            </label-switched-path>
            """.format(color, element2.text)

        change_xml = """
           <protocols>
              <mpls>
                 <admin-groups>
                    <name>{0}</name>
                    <group-value>{1}</group-value>
                 </admin-groups>
                 {2}
              </mpls>
           </protocols>
        """.format(color, group_value, lsp_config).strip()

        jcs.emit_change(change_xml, "transient-change", "xml")

if __name__ == '__main__':
    main()
```

For more information about this example, see "Example: Configure Administrative Groups for LSPs" on page 684.

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **lsp-admin.xsl**, **lsp-admin.slax**, or **lsp-admin.py** as appropriate, and copy it to the **/var/db/scripts/commit** directory on the device.

   > **NOTE**: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

2. If the script is written in Python, enable the execution of unsigned Python scripts.

```
[edit]
user@host# set system scripts language python
```

   > **NOTE**: Configure the `language python3` statement to use Python 3 to execute Python scripts, or configure the `language python` statement to use Python 2.7 to execute Python scripts. For more information, see *language*.

3. Select the following test configuration stanzas, and press Ctrl+c to copy them to the clipboard. If you are using the SLAX or Python version of the script, update the filename at the `[edit system scripts commit file]` hierarchy level.

```
system {
    scripts {
        commit {
            allow-transients;
            file 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;
        }
    }
}
```

4. 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

5. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying Script Execution

Purpose

Verify that the script behaves as expected.

Action

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;
}
```

RELATED DOCUMENTATION

Overview of Creating Custom Configuration Syntax with Commit Script Macros | 632
Create Custom Configuration Syntax with Commit Script Macros | 633

CHAPTER 26

Commit Script Examples

IN THIS CHAPTER

Example: Adding a Final then accept Term to a Firewall

IN THIS SECTION

This commit script example adds a `then accept` statement to any firewall filter that does not already end with an explicit `then accept` statement.

Requirements

This example uses a device running Junos OS.

Overview and Commit Script

Each firewall filter in 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. In this example, the commit script adds a final `then accept` statement to any firewall filter that does not already end with an explicit `then accept` statement.

The example script is shown in both XSLT and SLAX 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: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>;
                    }
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **add-accept.xsl** or **add-accept.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **add-accept.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command. The script requires that all firewall filters end with an explicit `then accept` statement. The sample configuration stanzas include the `test` filter with two terms but do not include an explicit `then accept` statement. When you issue the `commit` command, the script adds the missing `then accept` statement and commits the configuration. When you 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
```

In configuration mode, issue the `show firewall` command to review the modified configuration. 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: Adding T1 Interfaces to a RIP Group

IN THIS SECTION

This example shows how to use commit scripts to decrease the amount of manual configuration, specifically how to add every T1 interface configured at the `[edit interfaces]` hierarchy level to the `[edit protocols rip group test]` hierarchy level.

Requirements

This example uses a device running Junos OS with T1 interfaces.

Overview and Commit Script

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 how to use commit scripts to add 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.

The example script is shown in both XSLT and SLAX 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: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;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **rip-t1.xsl** or **rip-t1.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **rip-t1.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the script behaves as expected.

Action

Issue the `show protocols rip group test` command. All T1 interfaces should now appear under the `[edit protocols rip group test]` hierarchy level.

```
[edit]
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: Assign a Classifier Using a Commit Script

IN THIS SECTION

For each interface configured with the IPv4 protocol family, this commit script automatically assigns a specified classifier, which associates incoming packets with a forwarding class and loss priority as well as assigns packets to an output queue.

Requirements

This example uses a device running Junos OS.

Overview and Commit Script

In the 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`. The `fc-q3` classifier must be configured at the `[edit class-of-service]` hierarchy level.

The example script is shown in both XSLT and SLAX 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: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";
                        }
                    }
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **classifier.xsl** or **classifier.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **classifier.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

b. Press Enter.

c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command. In the test configuration stanzas, the fe-0/0/0.0 interface is configured with the `family inet` statement. Because the interface is configured with the IPv4 protocol family, the script automatically assigns the `fc-q3` classifier to the interface, which is indicated in the `commit` command output.

```
[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
```

View the configuration to verify that the script-generated changes are present. Issue the `show class-of-service` configuration mode command. The output 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: Automatically Configure Logical Interfaces and IP Addresses

IN THIS SECTION

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 and macro to complete the task quickly and with no errors.

Requirements

This example uses a device running Junos OS with physical ATM interfaces.

Overview and Commit Script

The following commit script 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.

The example script is shown in both XSLT and SLAX 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/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;
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and run the script:

1. Copy the script into a text file, name the file **atm-logical.xsl** or **atm-logical.slax** as appropriate, and download it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **atm-logical.slax**.

```
system {
    scripts {
        commit {
            allow-transients;
```

```
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Issue the `commit` command to commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the correct changes are integrated into the configuration.

Action

Before you commit the configuration, you can verify that the commit script will produce the correct results by issuing the `show interfaces at-1/2/3 | display commit-scripts` configuration mode command. After you commit the configuration, you can review the active configuration by issuing the `show configuration interfaces at-1/2/3` operational mode command. The following output appears:

```
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;
    }
}
```

Meaning

The `| display commit-scripts` option displays the configuration data after all commit scripts have been applied. The output includes both persistent and transient changes. If the appropriate `unit` and `vci` are configured on each ATM interface, the commit script executes successfully during a commit operation. After you commit the configuration, you can review the active configuration by issuing the `show configuration interfaces at-1/2/3` operational mode command.

Example: Configure Administrative Groups for LSPs

IN THIS SECTION

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.

This commit script example searches for `apply-macro` statements with the `color` parameter included at the `[edit protocols mpls]` hierarchy level. For each `apply-macro` statement, the script uses the data provided to generate a *transient change* and expand the macro into a standard Junos OS administrative group for LSPs.

Requirements

This example uses a device running Junos OS.

Overview and Commit Script

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 a line-by-line explanation of this script, see "Example: Creating Custom Configuration Syntax with Commit Script Macros" on page 644.

The example script is shown in both XSLT and SLAX 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: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;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **lsp-admin.xsl** or **lsp-admin.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **lsp-admin.slax**.

```
system {
    scripts {
        commit {
            allow-transients;
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the script behaves as expected.

Action

Issue the `show protocols mpls | display commit-scripts` configuration mode command and review the output. Adding the `| display commit-scripts` option allows you to see the configuration statements that are generated by transient changes.

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: Configure a Default Encapsulation Type

IN THIS SECTION

This commit script example configures default Cisco HDLC encapsulation on SONET/SDH interfaces not configured as aggregate interfaces.

Requirements

This example uses a device running Junos OS with SONET/SDH interfaces.

Overview and Commit Script

Point-to-Point Protocol (PPP) encapsulation is the default encapsulation type for physical interfaces. You do not need to 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 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";
            }
        }
```

```
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **so-encap.xsl** or **so-encap.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **so-encap.slax**.

```
system {
    scripts {
        commit {
            allow-transients;
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the script behaves as expected.

Action

When you issue the `commit` command, the 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: Configure Dual Routing Engines

IN THIS SECTION

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.

Requirements

This example uses a device running Junos OS with dual Routing Engines.

Overview and Commit Script

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, **dual-re.xsl**, generates 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, **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 generates 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.

The example scripts are shown in both XSLT and SLAX syntax:

XSLT Syntax: 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: 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: 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: 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]";
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and run the scripts:

1. Copy the scripts into two text files, name the files **dual-re.xsl** and **dual-re2.xsl** or **dual-re.slax** and **dual-re2.slax** as appropriate, and copy them to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **dual-re.slax** and **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 dual-re.xsl;
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Commit Script Changes

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command. 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: Configure an Interior Gateway Protocol on an Interface

IN THIS SECTION

This commit script example uses a macro to automatically include an interface at the `[edit protocols]` hierarchy level and to configure the proper interior gateway protocol (IGP) on the interface.

Requirements

This example uses a device running Junos OS.

Overview and Commit Script

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 commit script and macro to automatically include the interface at the `[edit protocols]` hierarchy level and to configure the proper 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.

The example script is shown in both XSLT and SLAX 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/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;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **if-class.xsl** or **if-class.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **if-class.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the script behaves as expected.

Action

View the configuration to verify that the manual changes and the script-generated changes are present.

When you issue the `show interfaces` configuration mode command, the changes added by the sample configuration stanzas should be present in the configuration.

```
[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;
        }
    }
}
```

When you issue the `show protocols` configuration mode command, the script-generated changes should be present in the configuration.

```
[edit]
user@host# show protocols
isis {
    interface t3-0/0/0.0;
}
ospf {
    area 10.4.0.0 {
        interface so-1/2/3.0;
    }
}
```

Example: Control IS-IS and MPLS Interfaces

IN THIS SECTION

This example shows how to use commit scripts to decrease the amount of manual configuration.

Requirements

This example uses a device running Junos OS.

Overview and Commit Script

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 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.

The example script is shown in both XSLT and SLAX 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: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;
                    }
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **iso.xsl** or **iso.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **iso.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command.

```
[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
```

Issue the `show interfaces` command. Confirm that the loopback interface is not altered and that 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: Control LDP Configuration

IN THIS SECTION

This commit script example generates a warning on LDP-enabled devices for any interfaces that are configured at either the `[edit protocols ospf]` or `[edit protocols isis]` hierarchy level but are not configured at the `[edit protocols ldp]` hierarchy level. A second test ensures that all LDP-enabled interfaces are configured for an interior gateway protocol (IGP). The example also provides instructions for excluding a particular interface from the commit script LDP test.

Requirements

This example uses a router running Junos OS.

Overview and Commit Script

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 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 generated 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 generated.

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 generated.

The example script is shown in both XSLT and SLAX 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: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";
                    }
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **ldp.xsl** or **ldp.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **ldp.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Script Execution

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command. The sample configuration stanzas enable LDP on the device and configure the so-1/2/2 and so-1/2/3 interfaces at the `[edit protocols isis]` hierarchy level and the ge-3/2/1 and ge-2/2/1 interfaces at the `[edit protocols ospf]` hierarchy level.

Because ge-2/2/1 is also configured at the `[edit protocols ldp]` hierarchy level, the script does not issue a warning message for this interface during the commit operation. The configuration includes the `apply-macro no-ldp` statement under the so-1/2/2 interface, so the script does not test this interface or issue a warning message for it, even though it is not configured at the `[edit protocols ldp]` hierarchy.

Neither so-1/2/3 nor ge-3/2/1 is configured at the `[edit protocols ldp]` hierarchy level as required by the commit script, so a warning is issued for both interfaces. The ge-1/2/1 interface is configured at the `[edit protocols ldp]` hierarchy. However, it is not configured for an IGP, so the commit script also issues a warning for the ge-1/2/1 interface.

```
[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: Create a Complex Configuration Based on a Simple Interface Configuration

IN THIS SECTION

This commit script example uses a macro to automatically expand a simple interface configuration.

Requirements

This example uses a device running Junos OS.

Overview and Commit Script

IN THIS SECTION

This example uses a commit script and 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 the 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.

The example script is shown in both XSLT and SLAX 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: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;
                    }
                }
            }
        }
    }
}
```

Topology

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **if-params.xsl** or **if-params.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **if-params.slax**.

```
system {
    scripts {
        commit {
            allow-transients;
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the script behaves as expected.

Action

Issue the `show interfaces | display commit-scripts | display inheritance` configuration mode command. The `| display commit-scripts` option displays all the statements that are in the configuration, including statements that are generated by transient changes. The `| display inheritance` option displays inherited configuration data and information about the source group from which the configuration has been inherited. This option also shows interface ranges configuration data in expanded format and information about the source interface-range from which the configuration has been expanded. You should see the following output:

```
[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;
        }
    }
}
```

Issue the `show protocols | display commit-scripts` configuration mode command. You should see the following output:

```
[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: Impose a Minimum MTU Setting

IN THIS SECTION

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. In this example, a commit script tests the MTU of SONET/SDH interfaces. If the MTU is less than a specified minimum value, the commit script reports the error and causes the commit operation to fail.

Requirements

This example uses a device running Junos OS with SONET/SDH interfaces.

Overview and Commit Script

This example tests the MTU of SONET/SDH interfaces, reports when the MTU is less than the value of the `min-mtu` parameter, here set to 2048, and causes the commit operation to fail. The `for` loop selects all SONET/SDH interfaces that start with so- and that have an MTU statement that is defined and less than the value of `min-mtu`. For the selected interfaces, the script generates an error, which includes the location of the interface in the configuration hierarchy and the MTU configured for that interface.

The example script is shown in both XSLT and SLAX 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: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 ".";
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **so-mtu.xsl** or **so-mtu.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **so-mtu.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Commit Script Output

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command. The sample configuration stanzas configure two SONET/SDH interfaces so-1/2/2 and so-1/2/3. The so-1/2/3 interface is configured with an MTU of 576, so the script generates an error message, and the commit operation fails. The following output appears after issuing a `commit` command:

```
[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: Limit the Number of ATM Virtual Circuits

IN THIS SECTION

This commit script example limits the number of *Asynchronous Transfer Mode* (ATM) virtual circuits (VCs) configured on an ATM interface.

Requirements

This example uses a device running Junos OS with an ATM interface.

Overview and Commit Script

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.

The example script is shown in both XSLT and SLAX 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: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.";
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **atm-vc-limit.xsl** or **atm-vc-limit.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **atm-vc-limit.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Commit Script Output

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command. The sample configuration stanzas configure 12 virtual circuits on the ATM interface atm-1/2/3. Because the commit script only allows 10 ATM VCs to be configured on any ATM interface, the script generates an error, and the commit operation fails. The following output appears after issuing a `commit` command:

```
[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: Limit the Number of E1 Interfaces

IN THIS SECTION

This commit script example limits the number of E1 interfaces configured on a Channelized STM1 Intelligent Queuing (IQ) *PIC* to avoid contention issues with per-unit-schedulers.

Requirements

This example uses a device running Junos OS with a Channelized STM1 Intelligent Queuing (IQ) PIC.

Overview and Commit Script

The following script ensures that there are no more than 16 E1 interfaces configured on a channelized STM1 IQ interface. 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 parameter `limit`. If there are more E1 interfaces than `limit`, a commit error is generated, and the commit operation fails.

The example script is shown in both XSLT and SLAX 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: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.";
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **e1-limit.xsl** or **e1-limit.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **e1-limit.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Commit Script Execution

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command. The sample configuration stanzas channelize a cstm1-0/1/0 interface into 17 E1 interfaces, so the script generates an error, and the commit operation fails. The following output appears after issuing a `commit` command:

```
[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: Load a Base Configuration

IN THIS SECTION

This commit script example sets up a sample base configuration on a device running Junos OS.

Requirements

This example uses a device running Junos OS.

Overview and Commit Script

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

The example script is shown in both XSLT and SLAX 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: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>
                $ABC123
            </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>
                $ABC123
            </secret>
        </radius-server>
        <radius-server>
            <name>192.168.4.240</name>
            <secret>
                $ABC123
            </secret>
        </radius-server>
        <login>
            <class>
                <permissions>all</permissions>
            </class>
            <user>
                <name>johnny</name>
                <uid>928</uid>
                <class>superuser</class>
                <authentication>
                    <encrypted-password>
                        $ABC123
                    </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>
                           "$ABC123";
        }
        <name-server> {
            <name> "192.168.5.68";
        }
        <name-server> {
            <name> "172.17.28.100";
        }
        <radius-server> {
            <name> "192.168.170.241";
            <secret> "$ABC123";
        }
        <radius-server> {
            <name> "192.168.4.240";
            <secret> "$ABC123";
        }
        <login> {
            <class> {
                <permissions> "all";
            }
            <user> {
                <name> "johnny";
                <uid> "928";
                <class> "superuser";
                <authentication> {
                    <encrypted-password>"$ABC123";
            }
        }
    }
    <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;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script 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 test 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 for 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the script behaves as expected.

Action

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
...
```

Example: Prepend a Global Policy

IN THIS SECTION

This commit script example ensures that a *BGP* global import policy is applied to all your BGP imports before any other import policies are applied.

Requirements

This example uses a device running Junos OS.

Overview and Commit Script

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 commit script 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 generated, 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.

The example script is shown in both XSLT and SLAX 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: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";
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **bgp-global-import.xsl** or **bgp-global-import.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **bgp-global-import.slax**.

```
system {
    scripts {
        commit {
            allow-transients;
            file 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 64500;
}
protocols {
    bgp {
        group fish {
            neighbor 192.168.16.4 {
                import [ blue green ];
                peer-as 64501;
            }
            neighbor 192.168.16.6 {
                peer-as 64502;
            }
        }
    }
}
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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Verify that the script behaves as expected.

Action

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 64501;
        }
        neighbor 192.168.16.6 {
            peer-as 64502;
        }
    }
}
```

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. Issue the `show protocols | display commit-scripts` to view all configuration statements including transient changes.

```
[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 64501;
        }
        neighbor 192.168.16.6 {
            peer-as 64502;
        }
```

```
    }
}
```

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. Issue the `show protocols | display commit-scripts` command to view all configuration statements including transient changes.

```
[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 64501;
        }
        neighbor 192.168.16.6 {
            import [ bgp_global_import green ];
            peer-as 64502;
        }
    }
}
```

Example: Prevent Import of the Full Routing Table

IN THIS SECTION

In the 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.

Requirements

This example uses a device running Junos OS.

Overview and Commit Script

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).

The example script is shown in both XSLT and SLAX 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: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'";
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **import.xsl** or **import.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **import.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

b. Press Enter.

c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying the Commit Script Execution

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command. The sample configuration configures an `import` statement at the `[edit protocols ospf]` hierarchy level. Because the policy contains a `then accept` term with no match conditions, the script generates an error, and the commit operation fails. The following output appears after issuing a `commit` command:

```
[edit]
user@host# commit
[edit protocols ospf]
    'import bad-news;'
    policy contains bare 'then accept'
error: 1 error reported by commit scripts
error: commit script failure
```

Example: Require Internal Clocking on T1 Interfaces

IN THIS SECTION

This example shows how to use a commit script to require that T1 interfaces be configured with internal clocking.

Requirements

This example uses a device running Junos OS with T1 interfaces.

Overview and Commit Script

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 generated, and the configuration is not committed.

The example script is shown in both XSLT and SLAX 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, '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.";
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **clocking-error.xsl** or **clocking-error.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **clocking-error.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying Commit Script Execution

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command. The sample configuration stanzas configure two T1 interfaces t1-0/0/0 and t1-0/0/1. Interface t1-0/0/0 is configured with the `clocking external` statement, and interface t1-0/0/1 does not include any `clocking` statement. The script generates an error, and the commit operation fails. The following output appears after issuing a `commit` command:

```
[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: Require and Restrict Configuration Statements

IN THIS SECTION

Junos OS commit scripts enforce custom configuration rules. When a *candidate configuration* is committed, it is inspected by each active commit script. This example uses a commit script to specify required and prohibited configuration statements.

Requirements

This example uses a device running Junos OS that has the Ethernet management interface fxp0.

Overview and Commit Script

This example shows how to use a commit script to specify required and prohibited configuration statements. The following 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 generates an error message, and the commit operation fails.

The example script is shown in both XSLT and SLAX 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: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"/> <!-- 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;
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **no-nukes.xsl** or **no-nukes.slax** as appropriate, and copy it to the **/var/db/scripts/commit/** directory on the device.

2. Select the following test 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 **no-nukes.slax**.

```
system {
    scripts {
        commit {
            file 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 by using the mouse and the paste icon.

   b. Press Enter.

   c. Press Ctrl+d.

4. Commit the configuration.

```
user@host# commit
```

Verification

IN THIS SECTION

Verifying Commit Script Execution

Purpose

Verify that the script behaves as expected.

Action

Review the output of the `commit` command. The script requires that the Ethernet management interface (fxp0) is configured and enabled and that the `bgp` statement is included at the `[edit protocols]` hierarchy level. The sample configuration stanzas include the fxp0 interface but disable it. In addition, the `bgp` statement is not configured at the `[edit protocols]` hierarchy level. When you run the script, it generates an error, and the commit operation fails. The following output appears after issuing a `commit` command:

```
[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
```

CHAPTER 27

Junos XML and XSLT Tag Elements Used in Commit Scripts

IN THIS CHAPTER

<change> (SLAX and XSLT)

IN THIS SECTION

Usage

```
<change>
  <!-- tag elements representing configuration statements to load -->
</change>
```

Description

In Junos OS commit scripts, 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.

Release Information

Statement introduced in Junos OS Release 7.4.

RELATED DOCUMENTATION

<transient-change> (SLAX and XSLT) | 799

Overview of Generating Persistent or Transient Configuration Changes Using Commit Scripts | 598

Generate a Persistent or Transient Configuration Change in SLAX and XSLT Commit Scripts | 604

Example: Generate Persistent and Transient Configuration Changes Using Commit Scripts | 617

Create Custom Configuration Syntax with Commit Script Macros | 633

<syslog> (Junos XML)

IN THIS SECTION

Usage

```
<syslog xmlns="namespace-URL"  xmlns:xnm="namespace-URL">
    <message>syslog-message </message>
</syslog>
```

Description

Indicate that the commit script should record the given message in the system log file.

Attributes

xmlns (Optional) 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 (Optional) 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 "Generate a Custom Warning, Error, or System Log Message in Commit Scripts" on page 572.

Release Information

Statement introduced in Junos OS Release 7.4.

<transient-change> (SLAX and XSLT)

IN THIS SECTION

Usage

```
<transient-change>
  <!-- tag elements representing configuration statements to load -->
</transient-change>
```

Description

In Junos OS commit scripts, 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.

Release Information

Statement introduced in Junos OS Release 7.4.

RELATED DOCUMENTATION

<change> (SLAX and XSLT) | 796

Overview of Generating Persistent or Transient Configuration Changes Using Commit Scripts | 598

Generate a Persistent or Transient Configuration Change in SLAX and XSLT Commit Scripts | 604

Example: Generate Persistent and Transient Configuration Changes Using Commit Scripts | 617

Create Custom Configuration Syntax with Commit Script Macros | 633

xnm:error (Junos XML)

IN THIS SECTION

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>
```

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 "Generate a Custom Warning, Error, or System Log Message in Commit Scripts" on page 572.

Release Information

Statement introduced in Junos OS Release 7.4.

RELATED DOCUMENTATION

xnm:warning (Junos XML) | 802

xnm:warning (Junos XML)

IN THIS SECTION

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>
```

Description

Indicate that the commit script has encountered a problem with the configuration and pass a warning message to the Junos OS CLI or Junos XML protocol client application. 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 "Generate a Custom Warning, Error, or System Log Message in Commit Scripts" on page 572

Release Information

Statement introduced in Junos OS Release 7.4.

RELATED DOCUMENTATION

xnm:error (Junos XML) | 800

CHAPTER 28

Troubleshoot Commit Scripts

IN THIS CHAPTER

Display Commit Script Output

Table 48 on page 805 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 807.

NOTE: Tracing commit script processing, including the **cscript.log** file, is not supported on the QFX3000-G QFabric system.

Table 48: 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 about the commit operation and commit script execution.	`commit | display detail`
Display the underlying Extensible Markup Language (XML) data.	`commit | display xml`

Table 48: Commit Script Configuration and Operational Mode Commands *(Continued)*

Task	Command
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. This is the configuration format that each commit script receives as input. 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`
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*

RELATED DOCUMENTATION

Trace Commit Script Processing on Devices Running Junos OS

IN THIS SECTION

Commit script tracing operations track 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, which include errors, warnings, progress messages, and script processing events, in the **/var/log/cscript.log** file 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.

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. Commit the configuration.

```
[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" on page 807. 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 49 on page 808 summarizes useful filtering commands that display selected portions of the `cscript.log` file.

Table 49: 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

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, the traceoptions `events` flag is turned on, regardless of the configuration settings, and only important events are logged. This includes errors, warnings, progress messages, and script processing events. 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 50 on page 812 describes the meaning of the commit script tracing flags.

Table 50: Commit Script Tracing Flags

Flag	Description	Default Setting
all	Trace all commit script operations.	Off
events	Trace important commit script events, including errors, warnings, progress messages, and script processing 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

Trace Script Processing on Devices Running Junos OS Evolved

IN THIS SECTION

When you execute an interactive script, the script can generate output, including warnings and errors, in the CLI or RPC reply. When the system triggers non-interactive scripts, for example, when an event policy triggers an event script, the script does not direct output to the terminal. In either case, you might need more information about the execution of the script. Junos OS Evolved captures trace data for all applications by default. You can view the collected traces for additional script processing information, including the memory and CPU usage, script arguments, script execution, and warnings and errors.

Junos OS Evolved collects trace data from all applications on all nodes on the Routing Engine. Whereas Junos OS logs the trace data for each type of script in separate log files, Junos OS Evolved stores the trace data for all scripts in the same location. The trace log includes data for commit, event, op, and SNMP scripts; YANG action and translation scripts; and Juniper Extension Toolkit scripts.

How to Display Trace Data for Scripts

Junos OS Evolved stores the trace data from all nodes that is collected on the primary Routing Engine under the **/var/log/traces** directory. The `cscript` application handles scripts, and the trace data for scripts is stored under the ***node*.cscript.*sequence-number*** subdirectories.

To view trace data for scripts, issue the `show trace application cscript` operational mode command.

```
user@host> show trace application cscript
2021-05-20 09:11:42.239695672 re0:cscript:4176:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg =
"Process's current softlimit [134217728] hardlimit [-1]"
2021-05-20 09:11:42.239773157 re0:cscript:4176:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg =
"Process's limits are already set by parent process"
2021-05-20 09:11:42.239812430 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"op script processing begins"
2021-05-20 09:11:42.239855140 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: /usr/libexec/ui/cscript"
2021-05-20 09:11:42.239865140 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -mop"
2021-05-20 09:11:42.239866196 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -p"
2021-05-20 09:11:42.239867156 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: /"
2021-05-20 09:11:42.239868116 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -Q2"
```

```
2021-05-20 09:11:42.239869131 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -f"
2021-05-20 09:11:42.239882048 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: hello.py"
2021-05-20 09:11:42.239883202 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -d134217728"
2021-05-20 09:11:42.239884135 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -E"
2021-05-20 09:11:42.239885131 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg:  user admin logname admin host host tty /dev/pts/0 agent op-script current-
directory /var/home/admin pid 32212 ppid 32206"
2021-05-20 09:11:42.239886175 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -u"
2021-05-20 09:11:42.239887176 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: admin"
2021-05-20 09:11:42.239888251 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -U"
2021-05-20 09:11:42.239889287 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -i9"
2021-05-20 09:11:42.245988806 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"running op script 'hello.py'"
2021-05-20 09:11:42.246006519 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"opening op script '/var/db/scripts/op/hello.py'"
...
```

You can include the `terse` option to display just the timestamp and message.

```
user@host> show trace application cscript terse
2021-05-20 09:11:42.239695672 msg = "Process's current softlimit [134217728] hardlimit [-1]"
2021-05-20 09:11:42.239773157 msg = "Process's limits are already set by parent process"
2021-05-20 09:11:42.239812430 msg = "op script processing begins"
...
```

You can also refine the traces to display by specifying the trace time elapsed, process ID, and node. For example, the following command shows trace data for a specific process ID.

```
user@host> show trace application cscript pid 10683
2021-05-24 09:42:09.552687492 re0:cscript:10683:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg
= "Process's current softlimit [134217728] hardlimit [-1]"
2021-05-24 09:42:09.552819712 re0:cscript:10683:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg
= "Process's limits are already set by parent process"
```

```
2021-05-24 09:42:09.552897412 re0:cscript:10683:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg
= "action script processing begins"
2021-05-24 09:42:09.553025992 re0:cscript:10683:TRACE_INFO
CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1  msg = "arg: /usr/libexec/ui/cscript"
2021-05-24 09:42:09.553095062 re0:cscript:10683:TRACE_INFO
CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1  msg = "arg: -maction"
...
```

How to Modify Trace Settings for Scripts

Junos OS Evolved traces script processing by default and traces all applications at the info level for informational messages. You can configure trace settings for specific applications at the `[edit system application]` hierarchy level. For example, you can specify the trace level of the application on a given node.

To modify script tracing operations, configure the settings under the `[edit system trace application cscript]` hierarchy level. The following example configures the `cscript` application to trace script processing on node re0 at the debug level:

```
[edit]
user@host# set system trace application cscript node re0 level debug
user@host# commit
```

For more information about configuring trace settings, see *trace*.

Troubleshoot Commit Scripts

After you enable a commit script and issue a `commit` command, the commit script takes effect immediately.

Table 51 on page 816 describes some common problems that might occur.

Table 51: 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 commit script is present in the **/var/db/scripts/commit/** directory on your Junos 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:
	
	• Include the `allow-transients` statement at the `[edit system scripts commit]` or the `[edit system scripts commit file` *`filename`*`]` hierarchy level.
	
	• Remove the code that generates a transient change from the indicated script.
	
	• Disable the script in the configuration.

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.	• 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.
	• Make sure you have included the required boilerplate in your script. For more information, see "Required Boilerplate for Commit Scripts" on page 542.

Table 51: Troubleshooting Commit Scripts *(Continued)*

Problem	Solution
	• Make sure that the *XPath* expressions in the script contain valid configuration statements expressed as Junos XML protocol tag elements. You can verify the XML hierarchy by issuing the `show configuration | display xml` operational mode command.

	• 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.
	
	For 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)">`

The commit operation fails, and an error message in the system log file indicates: `Process (`*pid*`,cscript) attempted to exceed RLIMIT_DATA`	When committing a configuration that is inspected by one or more commit scripts, you might need to increase the amount of memory allocated to the commit scripts to accommodate the processing of large configurations. To increase the maximum memory allocated for each executed commit script, configure the `max-datasize` *size* statement with an appropriate memory limit in bytes at the `[edit system scripts commit]` hierarchy level before committing the configuration.

9

PART

Op Scripts

CHAPTER 29

Op Scripts Overview

IN THIS CHAPTER

Op Script Overview

IN THIS SECTION

Understanding Op Scripts

Junos 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 enable you to:

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

- Monitor the overall status of a device by periodically checking network warning parameters, such as high CPU usage.

Op 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. The Junos XML API is an XML representation of Junos configuration statements and operational mode commands. For additional information, see "Junos XML Management Protocol and Junos XML API Overview" on page 14.

Op scripts can be written in Python, Extensible Stylesheet Language Transformations (*XSLT*), or Stylesheet Language Alternative syntaX (SLAX). Op scripts use *XML Path Language* (XPath) to locate the operational objects to inspect and use automation script constructs to perform actions on the objects. The actions can change the configuration, modify the output, or execute additional commands based on the output.

You can invoke op scripts in a number of ways. For example, you can:

- Invoke an op script manually in the CLI.
- Invoke an op script automatically upon user login.
- Call an op script from another script.
- Invoke an op script through an API call.

When you invoke an op script, the Junos OS management process (*mgd*) executes the script.

Benefits of Op Scripts

- Mitigate human error by performing controlled configuration changes
- Shorten troubleshooting time and speed time to resolution for network issues
- Streamline tasks
- Enable customization of operational commands and their output

RELATED DOCUMENTATION

How Op Scripts Work | 821
Understanding Python Automation Scripts for Junos Devices | 271

How Op Scripts Work

Op scripts execute Junos OS operational commands and inspect the resulting output. After inspection, op scripts can manipulate the output or automatically correct errors within the device running Junos OS based on this output.

You enable op scripts by listing the filenames of one or more op script files within the `[edit system scripts op]` hierarchy level. To execute local op scripts, you must add the files to the appropriate op script directory on the device. For more information about op script file directories, see "Storing and Enabling Scripts" on page 448. Once added to the device, op scripts are invoked from the command line, using the `op` *`filename`* command.

You can also store and execute op scripts from a remote site. Remote op scripts are invoked from the command line using the `op url` *`url`* command. For more information about executing remote op scripts, see "Executing an Op Script from a Remote Site" on page 853.

You can use op scripts to generate changes to the device 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.

RELATED DOCUMENTATION

Op Script Overview | **819**

Execute an Op Script on the Local Device | **851**

Execute an Op Script from a Remote Site | **853**

CHAPTER 30

Create and Execute Op Scripts

IN THIS CHAPTER

Required Boilerplate for Op Scripts

SUMMARY

Define the boilerplate for op scripts.

Junos OS op scripts can be written in Extensible Stylesheet Language Transformations (*XSLT*), Stylesheet Language Alternative syntaX (SLAX), or Python. Op scripts must include the necessary boilerplate required for that script language for both basic script functionality as well as any optional functionality used within the script such as the Junos OS extension functions and named templates. This topic provides standard boilerplate that can be used in XSLT, SLAX, and Python op scripts.

SLAX and XSLT 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.

XSLT Boilerplate for Op Scripts

The XSLT op script boilerplate is as follows:

```
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 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 "Understanding Named Templates in Junos OS Automation Scripts" on page 427 .

```
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 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 22.

```
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 SLAX op script boilerplate is as follows:

```
version 1.2;

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

        */
    }
}
```

Python Boilerplate for Op Scripts

Python op scripts do not have a required boilerplate, but they must import any objects that are used in the script. Python op scripts can import the following:

- `Junos_Context` dictionary—Contains information about the script execution environment.
- `jcs` library—Enables the script to use Junos OS extension functions and Junos OS named template functionality in the script.
- `jnpr.junos` module and classes—Enables the script to use Junos PyEZ.

For example:

```
from junos import Junos_Context
from jnpr.junos import Device
import jcs
```

```
if __name__ == '__main__':
```

Python automation scripts do not need to include an interpreter directive line (`#!/usr/bin/env python`) at the start of the script. However, the program will still execute correctly if one is present.

RELATED DOCUMENTATION

Understanding Extension Functions in Junos OS Automation Scripts | 337

Understanding Named Templates in Junos OS Automation Scripts | 427

Global Parameters and Variables in Junos OS Automation Scripts | 327

Map Operational Mode Commands and Output Fields to Junos XML Notation

In SLAX and XSLT op scripts, you use tag elements from the Junos XML *API* to represent operational mode commands and output fields in the scripts. For the Junos XML equivalent of commands and output fields, consult the *Junos XML API Operational Developer 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>
                ...
```

RELATED DOCUMENTATION

How to Use RPCs and Operational Mode Commands in Op Scripts

IN THIS SECTION

Most Junos OS operational mode commands have XML equivalents. Op scripts can execute these XML commands on a local or remote device using the *remote procedure call* (RPC) protocol. All operational mode commands that have XML equivalents are listed in the *Junos XML API Operational Developer Reference*.

Use of RPCs 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 a SLAX or XSLT op script, include the RPC in a variable declaration, and then invoke the RPC using the `jcs:invoke()` or `jcs:execute()` extension function with RPC variable as an argument. The `jcs:invoke()` function executes the RPC on the local device. The `jcs:execute()` function, in conjunction with a connection handle, executes the RPC on a remote device.

The following snippet, which invokes an RPC on the local device, is expanded and fully described in "Example: Customize Output of the show interfaces terse Command Using an Op Script" on page 877:

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);
```

The following snippet invokes the same RPC on a remote device:

XSLT Syntax

```
<xsl:variable name="rpc">
    <get-interface-information/> # Junos RPC for the show interfaces command
</xsl:variable>
<xsl:variable name="connection" select="jcs:open('198.51.100.1', 'bsmith', 'test123')"/>
<xsl:variable name="out" select="jcs:execute($connection, $rpc)"/>
<xsl:value-of select="jcs:close($connection)"/>
...
```

SLAX Syntax

```
var $rpc = <get-interface-information>;
var $connection = jcs:open('198.51.100.1', 'bsmith', 'test123');
var $out = jcs:execute($connection, $rpc);
expr jcs:close($connection);
```

In Python op scripts, RPCs are easy to execute using Junos PyEZ APIs. Each instance of the Junos PyEZ `Device` class has an `rpc` property that enables you to execute any RPC available through the Junos XML API. After establishing a session with a local or remote device, you can execute the RPC by appending the `rpc` property and RPC method name to the device instance. The return value is an XML object starting at the first element under the `<rpc-reply>` tag.

To execute the RPC on the local device, create the `Device` instance using an empty argument list. To execute the RPC on a remote device, create an instance of `Device` using the appropriate arguments to connect to that device.

The following code invokes an RPC on the local device and prints the reply:

Python Syntax

```
from jnpr.junos import Device
from lxml import etree

with Device() as jdev:
    rsp = jdev.rpc.get_interface_information()
    print (etree.tostring(rsp, encoding='unicode'))
```

The following code invokes the same RPC on a remote device and prints the reply:

Python Syntax

```
from jnpr.junos import Device
from lxml import etree
import jcs

user = jcs.get_input('Enter username: ')
password = jcs.get_secret('Enter user password: ')

with Device(host='198.51.100.1', user=user, password=password) as jdev:
    rsp = jdev.rpc.get_interface_information()
    print (etree.tostring(rsp, encoding='unicode'))
```

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 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 (198.51.100.1)' 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 'router2,198.51.100.1' (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 (198.51.100.1)' 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 'router2,198.51.100.1' (RSA) to the list of known hosts.
```

Displaying the RPC Tags for a Command

You can display the RPC XML tags for operational mode commands in the CLI of the device. To display the RPC XML tags for a 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>
```

SLAX and XSLT scripts can execute RPCs using the RPC XML tags. Python scripts must convert the RPC tags and command options into a format suitable for Python. For more information about using Junos PyEZ to execute RPCs and about mapping RPC tags to the corresponding Python method and method arguments, see Using Junos PyEZ to Execute RPCs on Devices Running Junos OS.

Using Operational Mode Commands in Op Scripts

Some operational mode commands do not have XML equivalents. SLAX and XSLT scripts can execute commands that have no XML equivalent using the `<command>` element. Python scripts can execute these commands by using the Junos PyEZ `cli()` method defined in the `Device` class.

If a command is not listed in the *Junos XML API Operational Developer 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, for example:

```
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 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 similar to 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 Developer Reference* is usually more reliable.

SLAX and XSLT op scripts 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: Display DNS Hostname Information Using an Op Script" on page 892.

```
<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>
...
```

Python op scripts can execute commands that have no XML equivalent by using Junos PyEZ APIs. The `cli()` method defined in the `Device` class executes an operational mode command and returns the output in text format. For example:

```
from jnpr.junos import Device

def main():

    with Device() as jdev:
        res = jdev.cli('show host hostname', warning=False)
        print (res)

if __name__ == "__main__":
    main()
```

You can also specify `format='xml'` to return the output formatted as Junos OS XML elements. For more information about the Junos PyEZ `cli()` method, see http://junos-pyez.readthedocs.org/en/latest/_modules/jnpr/junos/device.html#Device.cli .

RELATED DOCUMENTATION

open() Function (SLAX and XSLT) | 394

execute() Function (SLAX and XSLT) | 370

Map Operational Mode Commands and Output Fields to Junos XML Notation | 826

Declare and Use Command-Line Arguments in Op Scripts | 833

Junos PyEZ

Declare and Use Command-Line Arguments in Op Scripts

IN THIS SECTION

Junos OS op scripts can accept command-line arguments when you invoke the script. You can include declarations in the op script or statements in the configuration that enable a user to see the list of possible arguments when they request context-sensitive help for the op script in the CLI. The script must also include any necessary declarations and code to process those arguments. The following sections detail how to define the arguments and help text and use the arguments in an op script.

Declaring Op Script Command-Line Arguments

IN THIS SECTION

There are two ways to define the list of expected op scripts arguments that will be displayed when using context-sensitive help in the CLI:

- Include declarations in the op script
- Include statements in the Junos OS configuration

Script-generated and *configuration-generated* arguments have the same operational impact. The following sections explain how to use the different methods to define the op script arguments and display them in the CLI:

How to Define Arguments in the Op Script

You can declare an op script's expected command-line arguments directly in the Python, SLAX, or XSLT op script.

To declare command-line arguments in Python op scripts:

1. Declare a global dictionary named `arguments`.
2. For each argument, define a name-value pair that maps to the argument name and argument help text.

Python Syntax

```
# Define arguments dictionary
arguments = {'name1': 'description1', 'name2': 'description2'}
```

```
if __name__ == '__main__':
    ...
```

NOTE: To display the arguments in the CLI, Python scripts must include the `if __name__ == '__main__':` statement.

To declare command-line arguments in SLAX or XSLT op scripts:

1. Declare a global variable named `arguments`.
2. For each argument, define an `<argument>` element.
3. Within each `<argument>` element:
   - Define the `<name>` element with the name of the argument.
   - Optionally define a `<description>` element that provides the help text for that argument.

XSLT Syntax

```
<xsl:variable name="arguments">
    <argument>
        <name>name</name>
        <description>descriptive-text</description>
    </argument>
</xsl:variable>
```

SLAX Syntax

```
var $arguments = {
    <argument> {
        <name> "name";
        <description> "descriptive-text";
    }
}
```

How to Define Arguments in the Junos OS Configuration

You can declare an op script's expected command-line arguments in the Junos OS configuration, as an alternative to declaring the arguments directly in the op script.

To declare command-line arguments in the configuration:

1. Navigate to the `arguments` statement at the `[edit system scripts op file filename]` hierarchy level for the given script.
2. Configure the argument name.
3. Optionally configure the `description` statement to provide the help text for the argument.

For example:

```
[edit system scripts op op file file filename]
arguments {
    argument-name {
        description descriptive-text;
    }
}
```

How to Display Arguments in the Context-Sensitive Help

After you declare arguments in either the op script or the configuration, you can use the CLI's context-sensitive help to list the op script arguments. If you include the optional argument description, the CLI displays the help text with the argument name.

```
user@host> op filename ?
Possible completions:
  argument-name        description
  argument-name        description
```

You can also create a hidden argument for an op script by not including the argument declaration in the op script or the configuration. You use the argument as you normally would in the script, but the CLI does not display the argument or help text when you request context-sensitive help for that op script.

NOTE: If you configure command-line arguments in the Junos OS configuration and also declare arguments directly in the op script, the arguments that you declare in the script are still available, but the CLI does not list them under `Possible completions` when you issue the `op filename ?` command. This occurs because the management (mgd) process populates the list by first checking the configuration for arguments. The mgd process checks the script for arguments only if no arguments are found in the configuration.

Thus, if you declare arguments in the configuration, any arguments declared in the script become hidden in the CLI.

For more information about configuring help text for op scripts, see "Configure Help Text for Op Scripts" on page 843.

Using Command-Line Arguments in Op Scripts

IN THIS SECTION

You execute local op scripts with the `op filename` command. To pass command-line arguments to the script, include each argument name and value when you execute the script.

```
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.

The following sections discuss how to use the command-line arguments that are passed to Python, SLAX, and XSLT op scripts:

How to Use Arguments in Python Op Scripts

Python op scripts can use standard command-line parsing libraries to process and use command-line arguments. For example, you can use the Python `argparse` library to easily define required and optional arguments, specify default values, and handle the arguments in the script.

To enable users to more easily use the standard Python libraries to parse command-line arguments, we modified the way that the arguments are passed to Python op scripts. Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes command-line arguments to a Python op script, it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names. In earlier releases, the devices prefixes a single hyphen (-) to all argument names. You should ensure that your op script properly handles the arguments for your specific release.

The following examples use the `argparse` module to handle the script arguments. The examples define the global `arguments` dictionary, and the dictionary keys are used to define the expected arguments for the parser. We provide two example scripts, which appropriately handle the arguments in the specified releases.

Python Syntax (Junos OS Release 21.2R1 or later)

```
# Junos OS Release 21.2R1 and later
import argparse

arguments = {'arg1': 'description1', 'arg2': 'description2', 's': 'short option'}

def main():
    parser = argparse.ArgumentParser(description='This is a demo script.')

    # Define the arguments accepted by parser
    #  which use the key names defined in the arguments dictionary
    for key in arguments:
        if len(key) == 1:
            parser.add_argument(('-' + key), required=True, help=arguments[key])
        else:
            parser.add_argument(('--' + key), required=True, help=arguments[key])
    args = parser.parse_args()

    # Extract the value
    print (args.arg1)
    print (args.arg2)
    print (args.s)

if __name__ == '__main__':
    main()
```

Python Syntax (Junos OS Release 21.1 and earlier)

```
# Junos OS Release 21.1 and earlier
import argparse

arguments = {'arg1': 'description1', 'arg2': 'description2', 's': 'short option'}

def main():
    parser = argparse.ArgumentParser(description='This is a demo script.')
```

```
    # Define the arguments accepted by parser
    #  which use the key names defined in the arguments dictionary
    for key in arguments:
        parser.add_argument(('-' + key), required=True, help=arguments[key])
    args = parser.parse_args()

    # Extract the value
    print (args.arg1)
    print (args.arg2)
    print (args.s)

if __name__ == '__main__':
    main()
```

How to Use Arguments in SLAX and XSLT Op Scripts

To use command-line arguments in SLAX or XSLT op scripts, you must:

1. Include a parameter declaration for each argument
2. Ensure the parameter name is identical to the name that you defined in either the `arguments` variable declaration in the script or the `arguments` statement in the Junos OS configuration.

XSLT Syntax

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

SLAX Syntax

```
param $name;
```

The op script assigns the value for each script argument to the corresponding parameter, which can then be referenced throughout the script.

Example: Declaring Arguments in XSLT Op Scripts

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.

The following examples show how to declare the arguments in either the XSLT script or the configuration:

Declaring Arguments in the Op Script (script1)

```
<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)";
        }
    }
}
```

In addition to declaring the arguments in the script or the configuration, you must also declare the corresponding parameters in the script in order to reference the script arguments and access their values.

Declaring the Parameters

```
<xsl:param name="interface"/>
<xsl:param name="protocol"/>
```

Provide the argument names and values when you execute the script. For example:

Executing the Script

```
user@host> op script1 interface ge-0/2/0.0 protocol inet
```

Example: Declaring and Using Arguments in Python Op Scripts

Declare two arguments named `interface` and `p` in the Python op script. Execute the script, specifying the ge-0/2/0.0 interface and the `inet` protocol as values for the arguments. Select the appropriate argument handling statements based on your release. The script uses statements compatible with Junos OS Release 21.2R1 and later and comments out the statements for handling arguments in older releases.

Declaring Arguments in the Op Script (script1.py)

```
from jnpr.junos import Device
import argparse

# Define arguments dictionary
arguments = {'interface': 'Name of interface to display', 'p': 'Protocol to display (inet,
inet6)'}

def main():

    parser = argparse.ArgumentParser()

    # Argument handling for Junos OS Release 21.2R1 or later
    for key in arguments:
        if len(key) == 1:
            parser.add_argument(('-' + key), required=True, help=arguments[key])
        else:
            parser.add_argument(('--' + key), required=True, help=arguments[key])

    # Argument handling for Junos OS Release 21.1 and earlier
    #for key in arguments:
    #    parser.add_argument(('-' + key), required=True, help=arguments[key])

    args = parser.parse_args()

    try:
        with Device() as dev:

            res = dev.rpc.get_interface_information(
                interface_name=args.interface, terse=True, normalize=True)
```

```
            if (res.find("logical-interface/oper-status") is not None):
                print (args.interface + " status: " +
                    res.findtext("logical-interface/oper-status"))

            for elem in res.xpath("//address-family \
                [normalize-space(address-family-name)=$protocol]",
                protocol=args.p):
                if (elem.find("interface-address/ifa-local") is not None):
                    print ("inet address: " +
                    elem.find("interface-address/ifa-local").text)

    except Exception as err:
        print (err)

if __name__ == '__main__':
    main()
```

Alternatively, instead of including the `arguments` dictionary in the Python op script, you can include the arguments in the configuration exactly as you would for SLAX and XSLT scripts.

To view the op script arguments in the CLI's context-sensitive help, issue the `op` *`filename`* `?` command.

Displaying the Arguments

```
user@host> op script1.py ?
Possible completions:
  <[Enter]>            Execute this command
  <name>               Argument name
  detail               Display detailed output
  interface            Name of interface to display
  invoke-debugger      Invoke script in debugger mode
  p                    Protocol to display (inet, inet6)
  |                    Pipe through a command
```

Provide the argument names and values when you execute the script. For example:

Executing the Script

```
user@host> op script1.py interface ge-0/2/0.0 p inet
ge-0/2/0.0 status: up
inet address 198.51.100.1/24
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
21.2R1 and 21.2R1-EVO	Starting in Junos OS Release 21.2R1 and Junos OS Evolved Release 21.2R1, when the device passes command-line arguments to a Python op script, it prefixes a single hyphen (-) to single-character argument names and prefixes two hyphens (--) to multi-character argument names.

RELATED DOCUMENTATION

Configure Help Text for Op Scripts

IN THIS SECTION

You can provide help text to describe an op script and its arguments when the `?` is used to list possible completions in the *CLI*. To configure help text, 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 command-line 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
```

RELATED DOCUMENTATION

Declare and Use Command-Line Arguments in Op Scripts | 833
Map Operational Mode Commands and Output Fields to Junos XML Notation | 826
How to Use RPCs and Operational Mode Commands in Op Scripts | 827

Define Operational Mode Commands to Allow in an Op Script

Operation (op) scripts automate operational mode tasks and network troubleshooting on devices running Junos OS. Op scripts can execute operational mode commands within the script. By default, when a user executes a script, the system does not permit the user to execute operational mode commands within a script for which their login class does not normally have permission to execute. Starting in Junos OS Release 14.2, you can configure operational mode commands that a particular op script is allowed to execute. The permission to execute operational mode commands within a script applies to all users, and the commands that you specify are executed, even if the user who executes the script does not have permissions to execute the operational mode commands.

NOTE: Execution of configuration mode commands is not supported by this feature.

NOTE: The `allow-commands` statement is only supported for op scripts that are local to the device. Remote op scripts that are executed using the `op url` command do not support executing unauthorized operational mode commands even when you configure the `allow-commands` statement.

In the following example, the **sam.slax** script contains this code:

```
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> {
        var $usage = "This script updates the date on the device.";
        var $temp = jcs:output($usage);
        var $date = jcs:get-input("Enter the date and time (YYYYMMDDHHMM.ss): ");
        var $var = <command> "set date " _ $date;
        var $results = jcs:invoke( $var );
        copy-of $results;
    }
}
```

The op script **sam.slax** uses the `set date` operational mode command, which is not permitted for user user1, who has view permissions.

```
user1@device> op sam
This script updates the date on the device.
Enter the date and time (YYYYMMDDHHMM.ss): 201709111000.00
error: permission denied: date
```

To configure the **sam.slax** op script to execute the `set date` operational mode command (must be a user in the Junos OS `super-user` login class):

```
[edit system scripts op file sam.slax]
admin@device# set allow-commands date
admin@device# commit
```

User user1 can now successfully execute the op script.

```
user1@device> op sam
This script updates the date on the device.
Enter the date and time (YYYYMMDDHHMM.ss): 201709111000.00
Mon Sep 11 10:00:00 PDT 2017
```

To define the operational mode commands to allow in an op script:

1. Navigate to the op script where you want to allow operational mode commands.

   NOTE: Only users who belong to the Junos OS `super-user` login class can configure op scripts.

   ```
   [edit]
   admin@device# edit system scripts op file filename
   ```

2. Define the operational mode commands to allow.

   ```
   [edit system scripts op file filename]
   admin@device# set allow-commands "regular-expression"
   ```

3. Commit the configuration.

```
[edit system scripts op]
admin@device# commit
```

RELATED DOCUMENTATION

Enable an Op Script and Define a Script Alias | 847

Disable an Op Script | 855

Regular Expressions for Allowing and Denying Junos OS Operational Mode Commands, Configuration Statements, and Hierarchies

allow-commands

Enable an Op Script and Define a Script Alias

Operation (op) scripts are stored on a device's hard disk 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 "Store and Enable Junos Automation Scripts" on page 448 and "Store Scripts in Flash Memory" on page 451.

To prevent the execution of unauthorized Python code on devices running Junos OS, unsigned Python scripts must meet certain requirements before you can execute the scripts on a device. Starting in Junos OS Release 16.1R3, unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file. Prior to Junos OS Release 16.1R3, unsigned Python scripts must only be owned by the root user. For detailed information about the requirements for executing Python automation scripts on devices running Junos OS, see "Requirements for Executing Python Automation Scripts on Junos Devices" on page 276.

NOTE: If the device has dual Routing Engines and you want to enable an op script to execute on both Routing Engines, you can copy the script to the **/var/db/scripts/op** or **/config/scripts/op** directory on both Routing Engines, or you can issue the `commit synchronize scripts` command to synchronize the configuration and copy the scripts to the other Routing Engine as part of the commit operation.

You must enable an op script before it can be executed. To enable an op script, include the `file filename` statement at the `[edit system scripts op]` hierarchy level, and specify the name of the file containing the op script. Only users who belong to the Junos OS `super-user` login class can enable op scripts.

```
[edit system scripts op]
user@host# set file filename
```

SLAX and Python scripts must include the **.slax** or **.py** filename extension, respectively, in both the actual script name and the filename in the configuration. XSLT scripts do not require a filename extension, 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 file]` 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.

Optionally, you can define an alias for an op 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]
user@host# set command filename-alias
```

By default, you cannot execute unsigned Python scripts on devices running Junos OS. To enable the execution of unsigned Python automation scripts that meet the requirements outlined in "Requirements for Executing Python Automation Scripts on Junos Devices" on page 276, you must configure the `language python` or `language python3` statement at the `[edit system scripts]` hierarchy level.

```
[edit system scripts]
user@host# set language (python | python3)
```

To determine which op scripts are currently enabled on the device, use the `show` command to display the files configured at the `[edit system scripts op]` hierarchy level.

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

To ensure that the enabled files are on the device, list the contents of the **/var/run/scripts/op/** directory using the `file list /var/run/scripts/op` operational mode command.

```
user@host> file list /var/run/scripts/op
```

To execute the script, you can specify either the op script filename or the alias.

```
user@host> op (filename | filename-alias)
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
16.1R3	Starting in Junos OS Release 16.1R3, unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

RELATED DOCUMENTATION

Define Operational Mode Commands to Allow in an Op Script | 845
Disable an Op Script | 855

Configure Checksum Hashes for an Op Script

You can configure one or more checksum hashes that can be used to verify the integrity of a local 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.

Starting in Junos OS Release 18.2R2 and 18.3R1, Junos OS supports only the SHA-256 hash function for configuring script checksum hashes. Earlier releases support the *MD5*, *SHA-1*, and SHA-256 hash functions.

```
user@host> file checksum md5 /var/db/scripts/op/script1.slax
MD5 (/var/db/scripts/op/script1.slax) = 3af7884eb56e2d4489c2e49b26a39a97
```

```
user@host> file checksum sha1 /var/db/scripts/op/script1.slax
SHA1 (/var/db/scripts/op/script1.slax) = 00dc690fb08fb049577d012486c9a6dad34212c0
```

```
user@host> file checksum sha-256 /var/db/scripts/op/script1.slax
SHA256 (/var/db/scripts/op/script1.slax) =
150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
```

4. Configure the script and the `checksum` statement for 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, Junos OS recalculates the checksum value using the configured hash algorithm 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.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
18.3R1	Starting in Junos OS Release 18.2R2 and 18.3R1, Junos OS supports only the SHA-256 hash function for configuring script checksum hashes.

RELATED DOCUMENTATION

Execute an Op Script from a Remote Site | 853
Configure Checksum Hashes for a Commit Script | 564
Configuring Checksum Hashes for an Event Script | 1101
Configuring Checksum Hashes for an SNMP Script | 1130
checksum

Execute an Op Script on the Local Device

IN THIS SECTION

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` command in operational mode. You also can configure the device to execute an op script automatically when a member of a specific 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` command, and specify a script filename, a script alias as defined by the `command` statement at the `[edit system scripts op file *filename*]` hierarchy level, or a URL.

```
user@host> op (filename | filename-alias | url url)
```

For more information about executing op scripts from a remote site using `op url`, see "Execute an Op Script from a Remote Site" on page 853.

> **NOTE**: In order to execute Python op scripts from a remote site, you must configure the `allow-url-for-python` statement at the `[edit system scripts op]` hierarchy level.

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 *filename*` statement at the `[edit system login class *class-name*]` hierarchy level.

```
[edit system login]
class class-name {
    login-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 "Enable an Op Script and Define a Script Alias" on page 847).

```
[edit system login]
class super-user {
    login-script super-user-login.slax;
}
```

RELATED DOCUMENTATION

class (Defining Login Classes)

Execute an Op Script from a Remote Site | 853

Execute 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. You then execute a remote op script by issuing the `op` command and specifying the `url` *`url`* option. You can execute SLAX and XSLT op scripts from a remote site by default. To execute Python op scripts from a remote site, you must first configure the `allow-url-for-python` statement at the `[edit system scripts op]` hierarchy level. Because you cannot guarantee that scripts executed from remote sites are secure, we recommend that you only authorize trusted users to execute scripts using the `op url` command.

NOTE: Statements configured under the `[edit system scripts op]` hierarchy level are only enforced for op scripts that are local to the device. Thus, even if you configure memory allocation, script dampening, script start options, traceoptions, or other op script-specific statements within that hierarchy, the device does not apply the configuration when you execute a remote script using the `op url` command.

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.

   Starting in Junos OS Release 18.2R2 and 18.3R1, Junos OS supports only the SHA-256 hash function for script checksum hashes. Earlier releases support the *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. For Python scripts, configure the `allow-url-for-python` statement and the `language python` or `language python3` statement.

```
[edit]
user@host# set system scripts op allow-url-for-python
user@host# set system scripts language (python | python3)
user@host# commit
```

4. Place the script on the remote server.
5. Provide the script URL and the optional hash values to the administrators who will execute the script.
6. Execute the script by running the `op url` command and specifying the URL that points to the remote file.

```
user@host> op url https://www.juniper.net/scripts/script1.slax
key sha-256 150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
```

   This example shows how to include the `key` option and the SHA-256 checksum information.

If you instead want to prevent the execution of any op scripts from remote sites, configure the `no-allow-url` statement at the `[edit system scripts op]` hierarchy level.

```
user@host# set system scripts op no-allow-url
user@host# commit
```

When you configure the `no-allow-url` statement, issuing the `op url` *`url`* operational mode command generates an error. This statement takes precedence when the `allow-url-for-python` statement is also present in the configuration.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
18.3R1	Starting in Junos OS Release 18.2R2 and 18.3R1, Junos OS supports only the SHA-256 hash function for script checksum hashes.

RELATED DOCUMENTATION

Execute an Op Script on the Local Device | 851

Configure Checksum Hashes for an Op Script | 849

no-allow-url (Op Scripts)

allow-url-for-python (Op Scripts)

Disable an Op Script

You can disable an op script by deleting or deactivating the `file` *`filename`* statement at the `[edit system scripts op]` hierarchy in the configuration. To determine which op scripts are active on the device, issue the `show configuration system scripts op` operational mode command. The command output lists the enabled op scripts.

To delete an op script from the configuration:

1. From configuration mode in the CLI, enter the following command:

```
[edit]
user@host# delete system scripts op file filename
```

2. Commit the configuration.

```
user@host# commit
```

The `file` statement is removed from the configuration for the specified op script, and the `op` operational mode command no longer lists the op script filename as a valid completion.

To *deactivate* an op script in the configuration:

1. From configuration mode in the CLI, enter the following command:

```
[edit]
user@host# deactivate system scripts op file filename
```

2. Commit the configuration:

```
user@host# commit
```

The filename of the deactivated script remains in the configuration, but it is flagged with `inactive`. For example:

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

inactive: file script1.xsl;
file script2.xsl;
file script3.xsl;
```

NOTE: You can reactivate an op script using the `activate system scripts op file` *`filename`* command.

Alternatively, you can delete the script from the **/var/db/scripts/op** directory on a device's hard disk or from the **/config/scripts/op** directory on the flash drive. Only users in the Junos OS `super-user` login class can access and edit files in these directories.

If you delete a script, you should also remove the `file` statement at the `[edit system scripts op]` hierarchy level in the configuration. If you delete an op script, but the `file` statement remains in the configuration, the CLI lists this script as a valid completion for the `op` command, but Junos OS issues an invalid filename error when the script is executed.

If you deactivate or delete the `file` statement for an op script in the configuration, you must enable the script again in order to execute it.

RELATED DOCUMENTATION

Enable an Op Script and Define a Script Alias | 847
Define Operational Mode Commands to Allow in an Op Script | 845

CHAPTER 31

Op Script Examples

IN THIS CHAPTER

Change the Configuration Using SLAX and XSLT Scripts

IN THIS SECTION

SLAX and XSLT op and event scripts can use the `jcs:load-configuration` template to make structured changes to the Junos OS configuration. Experienced users, who are familiar with Junos OS, can write scripts that prompt for the relevant configuration information and modify the configuration accordingly.

This enables users who have less experience with Junos OS to safely modify the configuration using the script.

This topic discusses how to use the `jcs:load-configuration` template to modify the configuration.

jcs:load-configuration Template Overview

The `jcs:load-configuration` template is included in the **junos.xsl** import file. The template can:

- Load Junos XML configuration data into the candidate configuration using a `load merge`, `load replace`, or `load override` operation and commit the changes
- Roll back the active configuration to a previously committed configuration
- Load and commit the rescue configuration

When called, the `jcs:load-configuration` template performs the following actions on the target device:

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. The template call might fail if the configuration database is already locked or if there are existing, uncommitted changes in the candidate configuration when the template is called. If the template successfully loads the configuration data, but the commit fails, Junos OS discards the uncommitted changes when the database is unlocked.

The SLAX template syntax is:

```
call jcs:load-configuration($action="(merge | override | replace)",
  $commit-options=node-set, $configuration=configuration-data,
  $connection=connection-handle, $rescue="rescue", $rollback=number);
```

The XSLT 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:with-param name="rescue" select="&quot;rescue&quot;"/>
    <xsl:with-param name="rollback" select="number"/>
</xsl:call-template>
```

You provide arguments to the `jcs:load-configuration` template to specify:

- the connection handle to the device on which the changes will be made
- the changes to make to the configuration
- the load action that defines how to integrate the changes into the existing configuration
- optional commit options

You must establish a connection with the target device before calling the `jcs:load-configuration` template. To connect to a device, call the `jcs:open()` function with the necessary arguments. Then set the `jcs:load-configuration` `connection` parameter to the handle returned by the `jcs:open()` function.

The following sample code connects to the local device and modifies the configuration:

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

var $results := {
    call jcs:load-configuration($configuration=$config-changes, $connection=$conn);
}
var $close-results = jcs:close($conn);
```

When you call the `jcs:load-configuration` template, you can include the `configuration` parameter to load new configuration data on a device, you can specify the `rollback` parameter to revert the configuration to a previously committed configuration, or you can specify the `rescue` parameter to load and commit the rescue configuration.

Loading and Committing Configuration Data

SLAX and XSLT scripts can call the `jcs:load-configuration` template to modify the configuration. The `configuration` parameter defines the Junos XML configuration data to load, and the `action` parameter specifies how to load the data. The `commit-options` parameter defines the options to use during the commit operation.

The following sample op script calls the `jcs:load-configuration` template to modify the configuration to disable an interface. All of the values required for the `jcs:load-configuration` template are defined as variables, which are then passed into the template as arguments.

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.

SLAX syntax:

```
version 1.2;

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 $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> {
          <synchronize>;
          <log> "disabling interface " _ $interface;
      }
  }
  var $conn = jcs:open();

  var $results := {
      call jcs:load-configuration($action=$load-action, $commit-options=$options,
              $configuration=$config-changes, $connection=$conn);
  }
```

```
  if ($results//xnm:error) {
      for-each ($results//xnm:error) {
          <output> message;
      }
  }
  var $close-results = jcs:close($conn);

  }
}
```

For detailed information about this script, see "Example: Change the Configuration Using SLAX and XSLT Op Scripts" on page 864.

The equivalent XSLT code for the call to the `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)"/>
```

Loading and Committing the Rescue Configuration

A rescue configuration allows you to define a known working configuration or a configuration with a known state that you can restore at any time. SLAX and XSLT scripts can call the `jcs:load-configuration` template with the `rescue` parameter to load the rescue configuration, if one exists.

The following SLAX op script loads and commits the existing rescue configuration.

```
version 1.2;

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 "/var/db/scripts/import/junos.xsl";
```

```
match / {
    <op-script-results> {
        /* Open the connection */
        var $conn = jcs:open();

        /* Load and commit the rescue configuration */
        var $results = {
           call jcs:load-configuration($connection=$conn, $rescue="rescue");
        }
        expr jcs:output($results);

        /* Close the connection */
        expr jcs:close($conn);
    }
}
```

The equivalent XSLT script is:

```
<?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:import href="/var/db/scripts/import/junos.xsl"/>

  <xsl:template match="/">
    <op-script-results>

      <!-- Open the connection -->
      <xsl:variable name="conn" select="jcs:open()"/>

      <!-- Load and commit the rescue configuration -->
      <xsl:variable name="results">
        <xsl:call-template name="jcs:load-configuration">
          <xsl:with-param name="connection" select="$conn"/>
          <xsl:with-param name="rescue" select="&quot;rescue&quot;"/>
        </xsl:call-template>
      </xsl:variable>
      <xsl:value-of select="jcs:output($results)"/>

      <!-- Close the connection -->
```

```
      <xsl:value-of select="jcs:close($conn)"/>
    </op-script-results>

  </xsl:template>
</xsl:stylesheet>
```

Rolling Back the Configuration

SLAX and XSLT scripts can call the `jcs:load-configuration` template with the `rollback` parameter to revert the configuration to a previously committed configuration. The following SLAX op script prompts for the rollback number, and then loads the requested rollback configuration and commits it.

```
version 1.2;
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> {

        var $rollback_id = jcs:get-input("Rollback id: ");

        /* Open the connection */
        var $conn = jcs:open();

        /* Roll back the configuration and commit it */
        var $results = {
           call jcs:load-configuration($connection=$conn, $rollback=$rollback_id);
        }

        /* Close the connection */
        expr jcs:close($conn);
    }
}
```

```
user@host> op load-rollback
Rollback id: 1
```

RELATED DOCUMENTATION

jcs:load-configuration Template | 442
Example: Change the Configuration Using SLAX and XSLT Op Scripts | 864
Change the Configuration Using an Event Script | 1025

Example: Change the Configuration Using SLAX and XSLT Op Scripts

IN THIS SECTION

This example explains how to make structured changes to the Junos OS configuration using a SLAX *op script*.

Device Configuration

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **config-change.slax**, and copy it to the **/var/db/scripts/op/** directory on the device.

2. In configuration mode, configure the script's filename at the `[edit system scripts op file]` hierarchy level.

```
[edit system scripts op]
user@host# set file config-change.slax
```

3. Issue the `commit and-quit` command to commit the configuration and to return to operational mode.

```
[edit]
user@host# commit and-quit
```

4. Before running the script, issue the `show interfaces` *`interface-name`* operational mode command and record the current state of the interface that will be disabled by the script.

5. Execute the op script.

```
user@host> op config-change
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
```

Requirements

This example uses a device running Junos OS.

Overview and Op Script

SLAX and XSLT op scripts can use the `jcs:load-configuration` template, which is located in the **junos.xsl** import file, to make structured changes to the Junos OS configuration. This example creates a SLAX op script that uses 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.

In this example, the `usage` variable is initialized with a general description of the function of the script. When you run the script, it calls the `jcs:output()` function to output the usage description to the CLI. This enables you to verify that you are using the script for the correct purpose.

The script calls the `jcs:get-input()` function, which prompts for the name of the interface to disable, and stores the interface name in the `interface` variable. The `config-changes` variable stores the Junos XML configuration data to load on the device and references the `interface` variable. The `jcs:load-configuration` template call sets the value of the `configuration` parameter to the data stored in the `config-changes` variable.

The `load-action` variable is set to `merge`, which merges the new configuration data with the candidate configuration. This is the equivalent of the CLI configuration mode command `load merge`.

The `options` variable defines the options for the commit operation. It 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 for future reference.

The call to the `jcs:open()` function opens a connection with the Junos OS management process (mgd) on the local device and returns a connection handle that is stored in the `conn` variable. The script then calls the `jcs:load-configuration` template.

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 to the device. By default, the `jcs:load-configuration` template does not output messages to the CLI. This example searches for and prints `xmn:warning` and `xnm:error` messages in the response to quickly identify any issues with the commit.

SLAX Syntax

```
version 1.2;

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 specified interface." _
            "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 = jcs:open();

    var $results := {
      call jcs:load-configuration( $action=$load-action,
             $commit-options=$options,
             $configuration=$config-changes, $connection=$conn);
    }

    var $close-results = jcs:close($conn);

    if ($results//xnm:error) {
        for-each ($results//xnm:error) {
            <output> message;
        }
    }
    if ($results//xnm:warning) {
        for-each ($results//xnm:warning) {
            <output> message;
        }
    }

}
}
```

Verification

IN THIS SECTION

Verifying 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
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, this message might or might not appear in your log file.

```
user@host> show 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 mib2d[1434]: SNMP_TRAP_LINK_DOWN: ifIndex 526, ifAdminStatus down(2),
ifOperStatus down(2), ifName so-0/0/0
```

Verifying the Configuration Changes

Purpose

Verify that the correct changes are 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
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

jcs:load-configuration Template | 442

Change the Configuration Using SLAX and XSLT Scripts | 857

Example: Change the Configuration Using Python Op Scripts | 870

Example: Change the Configuration Using Python Op Scripts

IN THIS SECTION

Op scripts enable you to make controlled changes to the Junos OS configuration. Op scripts are advantageous, because they can gather operational information about a device and update the configuration based on that information. Experienced users who are familiar with Junos OS can write op scripts that prompt for the relevant configuration information and modify the configuration accordingly. This enables users who have less experience with Junos OS to safely modify the configuration using the script. This example demonstrates how to make changes to the Junos OS configuration using a Python *op script* that leverages Junos PyEZ APIs.

Requirements

This example uses the following hardware and software components:

- MX Series router running Junos OS Release 16.1R3 or later release that includes the Python extensions package.

Overview and Op Script

Python op scripts can make changes to the Junos OS configuration using the Junos PyEZ `jnpr.junos.utils.config.Config` utility. The Junos PyEZ `Config` utility provides instance methods to lock the configuration, load the configuration data and specify how to integrate it into the configuration, commit the configuration, and unlock the configuration. For more information about using Junos PyEZ to configure Junos devices, see Using Junos PyEZ to Configure Junos Devices. The Python op script in this example demonstrates how to update the configuration to disable an interface on the local device.

The Python op script imports the following:

- `Device` class—handles the connection to the Junos device
- `Config` class—performs configuration mode commands on the target device
- `jnpr.junos.exception` module—contains exceptions encountered when managing Junos devices
- `jcs` module—enables the script to execute supported extension functions

In this example, the `usage` variable is initialized with a general description of the script's function. When the script is executed, the script outputs the usage description on the CLI so that the user can verify the purpose for that script.

The script calls the `jcs.get_input()` extension function, which prompts the user to enter the name of the interface to disable, and stores the interface name in the `interface` variable. The `config_xml` variable is an XML string that defines the configuration changes.

The script does not supply a host parameter when creating the `Device` instance, which causes the `open()` method to establish a connection with the local device. This example creates the `Config` instance by using a context manager with `mode='exclusive'` to obtain an exclusive lock on the configuration while it's being modified. In this mode, the context manager automatically handles locking and unlocking the candidate configuration. The `Config` utility methods load the configuration changes into the candidate configuration as a `load merge` operation and commit the configuration. The `dev.close()` method closes the connection.

Python Script

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

def main():

    usage = """
```

```
        This script disables the specified interface.
        The script modifies the candidate configuration to disable
        the interface and commits the configuration to activate it.
    """
    print (usage)

    interface = jcs.get_input("Enter interface to disable: ")
    if not interface:
       print ("invalid interface")
       sys.exit(1)

    config_xml = """
        <configuration>
            <interfaces>
                <interface>
                    <name>{0}</name>
                    <disable/>
                </interface>
            </interfaces>
        </configuration>
    """.format(interface).strip()

    dev = Device()
    dev.open()
    try:
        with Config(dev, mode="exclusive") as cu:
            print ("Loading and committing configuration changes")
            cu.load(config_xml, format="xml", merge=True)
            cu.commit()

    except Exception as err:
        print (err)
        dev.close()
        return

    dev.close()

if __name__ == "__main__":
    main()
```

Configuration

IN THIS SECTION

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the script into a text file, name the file **config-change.py**, and copy it to the **/var/db/scripts/op/** directory on the device.

   > **NOTE**: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

2. In configuration mode, include the `file` **config-change.py** statement at the `[edit system scripts op]` hierarchy level.

```
[edit system scripts]
user@host# set op file config-change.py
```

3. Enable the execution of unsigned Python scripts on the device.

```
[edit system scripts]
user@host# set language python
```

   > **NOTE**: Configure the `language python3` statement to use Python 3 to execute Python scripts, or configure the `language python` statement to use Python 2.7 to execute Python scripts. For more information, see *language*.

4. Issue the `commit and-quit` command to commit the configuration and to return to operational mode.

```
[edit]
user@host# commit and-quit
```

5. Before running the script, issue the `show interfaces` *`interface-name`* operational mode command and record the current state of the interface that will be disabled by the script.

6. Execute the op script by issuing the `op config-change.py` operational mode command.

```
user@host> op config-change.py
    This script disables the specified interface.
    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
Loading and committing configuration changes
```

Verification

IN THIS SECTION

Verifying the Commit

Purpose

Verify that the *commit* succeeded.

Action

You should include code in your script that catches any warnings or errors associated with changing and committing the configuration. This enables 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.

```
user@host> show system commit
0   2010-09-22 17:08:17 PDT by user via netconf
```

- 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. Depending on your configuration settings for traceoptions, this message might or might not appear in your log file.

```
user@host> show log messages | last
Sep 22 17:08:13  host file[7319]: UI_COMMIT: User 'user' requested 'commit' operation
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
```

Verifying the Configuration Changes

Purpose

Verify that the correct changes are 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
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

Junos PyEZ

Example: Change the Configuration Using SLAX and XSLT Op Scripts | **864**

Store and Enable Junos Automation Scripts | **448**

Example: Customize Output of the show interfaces terse Command Using an Op Script

IN THIS SECTION

This example uses an *op script* to customize the output of the `show interfaces terse` command. A line-by-line explanation of the XSLT script is provided.

Requirements

This example uses a device running Junos OS.

Overview and 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
                                            fc00::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/10.0R1/junos">
    <interface-information xmlns="http://xml.juniper.net/junos/10.0R1/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. A line-by-line explanation of the script is provided.

```
<?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>interface</name>
```

```
            <description>Name of interface to display</description>
        </argument>
        <argument>
            <name>protocol</name>
            <description>Protocol to display (inet, inet6)</description>
        </argument>
    </xsl:variable>
    <xsl:param name="interface"/>
    <xsl:param name="protocol"/>
    <xsl:template match="/">
        <op-script-results>
            <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($rpc)"/>
            <interface-information junos:style="terse">
                <xsl:choose>
                    <xsl:when test="$protocol='inet' or $protocol='inet6'
                                 or $protocol='mpls' or $protocol='tnp'">
                        <xsl:for-each select="$out/physical-interface/
             logical-interface[address-family/address-family-name = $protocol]">
                            <xsl:call-template name="intf"/>
                        </xsl:for-each>
                    </xsl:when>
                    <xsl:when test="$protocol">
                        <xnm:error>
                            <message>
                                <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>
```

```
95              <admin-status><xsl:value-of select="$status"/></admin-status>
96              <xsl:choose>
97                  <xsl:when test="$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>
104         </logical-interface>
105     </xsl:template>
106 </xsl:stylesheet>
```

Line-by-Line Explanation

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 822.

```
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 generated.

```
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 78 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

The SLAX version of the script 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";

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;
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the XSLT or SLAX script into a text file, name the file **interface.xsl** or **interface.slax** as appropriate, and copy it to the **/var/db/scripts/op/** directory on the device.

2. In configuration mode, include the `file` statement at the `[edit system scripts op]` hierarchy level and **interface.xsl** or **interface.slax** as appropriate.

   ```
   [edit system scripts op]
   user@host# set file interface.(slax | xsl)
   ```

3. Issue the `commit and-quit` command to commit the configuration and to return to operational mode.

```
[edit]
user@host# commit and-quit
```

4. Execute the op script by issuing the `op interface` operational mode command.

Verification

IN THIS SECTION

Verifying the Op Script Output

Purpose

Verify that the script behaves as expected.

Action

Issue the `show interfaces terse` and `op interface` operational commands and compare the output. The `show interfaces terse` command displays the standard output. The `op interface` command displays the customized 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
                                            fc00::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 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
                                            fc00::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-->
```

Issue the `op interface` operational command for different hierarchy levels and review the output. For example:

```
user@host> op 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 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: Display DNS Hostname Information Using an Op Script

IN THIS SECTION

This example uses an *op script* to display *Domain Name System* (*DNS*) information for a device in your network.

Requirements

This example uses a device running Junos OS.

Overview and Op Script

This script displays 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 than using a *remote procedure call* (RPC).

The script is provided in two distinct versions, one using the `<xsl:choose>` element and the other using the `jcs:first-of()` function. Both versions accept the same argument and produce the same output. Each version is shown in both XSLT and SLAX syntax.

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;
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the XSLT or SLAX script into a text file, name the file **hostname.xsl** or **hostname.slax** as appropriate, and copy it to the **/var/db/scripts/op/** directory on the device.

2. In configuration mode, include the `file` statement at the `[edit system scripts op]` hierarchy level and **hostname.xsl** or **hostname.slax** as appropriate.

```
[edit system scripts op]
user@host# set file hostname.(slax | xsl)
```

3. Issue the `commit and-quit` command to commit the configuration and to return to operational mode.

```
[edit]
user@host# commit and-quit
```

4. Execute the op script by issuing the `op hostname <dns (`*`hostname`*` | `*`address`*`)>` operational mode command.

Verification

IN THIS SECTION

Verifying the Op Script Execution

Purpose

Verify that the script behaves as expected.

Action

When you issue the `op hostname` operational mode command without the `dns` option, DNS information is displayed for the local device:

```
user@host1> op hostname
Name:
host1 has address 10.168.71.246
```

When you issue the `op hostname dns` *hostname* command, DNS information is displayed for the specified device:

```
user@host1> op hostname dns router1
Name:
router1 has address 10.168.71.249
```

When you issue the `op hostname dns` *address* command, DNS information is displayed for the specified address:

```
user@host1> op hostname dns 10.168.71.249
Name:
249.71.168.10.IN-ADDR.ARPA domain name pointer router1
```

Example: Find LSPs to Multiple Destinations Using an Op Script

IN THIS SECTION

This example uses an *op script* to check for label-switched paths (LSPs) to multiple destinations.

Requirements

This example uses a device running Junos OS.

Overview and Op Script

The following example script, which is shown in both XSLT and SLAX, checks for LSPs to multiple destinations. The script takes one mandatory command-line argument, the address specifying the LSP endpoint. The address argument can include an optional prefix length. If no address is specified, the script generates an error message and halts execution.

The `get-configuration` variable stores the *remote procedure call* (RPC) to retrieve the `[edit protocols mpls]` hierarchy level of the device's committed configuration. This configuration is stored in the `config` variable. The `get-route-information` variable stores the RPC equivalent of the `show route` *`address`* `terse` operational mode command, where the value of the `destination` tag specifies *address*. The script sets this value to the address specified by the user on the command line. The script invokes the `get-route-information` RPC and stores the output in the `rpc-out` variable. If `rpc-out` does not contain any errors, the script examines all host route entries present at the `route-table/rt/rt-destination` node.

For each host route entry, if an LSP to the destination is configured in the active configuration, the script generates a "Found" message with the destination address and corresponding LSP name in the output. If an LSP to the destination is not configured, the output generates a "Missing" message containing the destination address and hostname.

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:import href="../import/junos.xsl"/>

   <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";

import "../import/junos.xsl";

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: ' _  .;
                    }
                }
            }
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the XSLT or SLAX script into a text file, name the file **lsp.xsl** or **lsp.slax** as appropriate, and copy it to the **/var/db/scripts/op/** directory on the device.

2. In configuration mode, include the `file` statement at the `[edit system scripts op]` hierarchy level and **lsp.xsl** or **lsp.slax** as appropriate.

```
[edit system scripts op]
user@host# set file lsp.(slax | xsl)
```

3. Issue the `commit and-quit` command to commit the configuration and to return to operational mode.

```
[edit]
user@host# commit and-quit
```

4. Execute the op script by issuing the `op lsp address` *address* operational mode command.

Verification

IN THIS SECTION

Verifying Script Execution

Purpose

Verify that the script behaves as expected.

Action

Issue the `op lsp address` *address* operational mode command to execute the script. The output varies depending on the configuration.

```
user@R4> op lsp address 10.168.215.0/24
Found: 192.168.215.1 (192.168.215.1) --> R4>R1
Found: 192.168.215.2 (192.168.215.2) --> R4>R2
Name: R3
Missing: 10.168.215.3
Name: R5
Missing: 10.168.215.4
```

```
Name: R6
Missing: 10.168.215.5
```

Example: Restart an FPC Using an Op Script

IN THIS SECTION

This example uses an *op script* to restart a *Flexible PIC Concentrator* (*FPC*).

Requirements

This example uses a device running Junos OS that contains a Flexible PIC Concentrator (FPC) or equivalent component.

Overview and Op Script

The following script, which is shown in both XSLT and SLAX formats, restarts an FPC given the slot number in which the FPC resides. The user provides the slot number in the command-line interface (CLI) when the script is invoked. The script stores the slot number as the value of the parameter `slot` and constructs the `request chassis fpc` command string to include the slot number of the FPC to restart. There is no Junos Extensible Markup Language (XML) equivalent for the `request chassis` commands. Therefore, this script invokes the `request chassis fpc` command directly rather that using a *remote procedure call* (RPC).

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.";
        }
    }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the XSLT or SLAX script into a text file, name the file **restart-fpc.xsl** or **restart-fpc.slax** as appropriate, and download it 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.

2. In configuration mode, include the `file` statement at the `[edit system scripts op]` hierarchy level and **restart-fpc.xsl** or **restart-fpc.slax** as appropriate.

```
[edit system scripts op]
user@host# set file restart-fpc.(slax | xsl)
```

3. Issue the `commit and-quit` command to commit the configuration and to return to operational mode.

```
[edit]
user@host# commit and-quit
```

4. Execute the op script by issuing the `op restart-fpc slot` *`slot-number`* operational mode command.

Verification

IN THIS SECTION

Verifying Op Script Execution

Purpose

Verify that the FPC has been restarted and is currently online.

Action

Execute the op script by issuing the `op` *`filename`* operational mode command. Supply the `slot` number of the FPC as an argument.

```
user@host> op restart-fpc slot 0
```

When you execute the script, you should see output similar to the following:

```
Restarting the FPC in slot 0. To verify, issue the "show chassis fpc" command.
```

Issue the `show chassis fpc detail` *`fpc-slot`* operational mode command.

```
user@host> show chassis fpc detail 0

Slot 0 information:
  State                                   Online
  Temperature                          36 degrees C / 96 degrees F
  Total CPU DRAM                     1024 MB
  Total RLDRAM                        256 MB
  Total DDR DRAM                     4096 MB
  Start time:                             2009-08-11 21:20:30 PDT
  Uptime:                                 0 hours, 1 minutes, 50 seconds
  Max Power Consumption                335 Watts
```

Meaning

The `show chassis fpc detail` command output displays the state, start time, uptime, and characteristics for the FPC. Verify that the FPC was restarted by checking the start time and uptime of the FPC. Verify the status of the restarted FPC by checking the state. If the status is `Present`, the FPC is coming up but is not yet online. If the status is `Online`, the FPC is online and running.

Example: Export Files Using an Op Script

IN THIS SECTION

The *op script* in this example uses the Junos XML protocol `file-put` operation to write to a file on a remote server and on the local device.

Requirements

This example uses a device running Junos OS.

Overview and Op Script

The Junos XML protocol `file-put` operation creates a file and writes the specified contents to that file. The basic syntax for using the `file-put` command is as follows:

```
<rpc>
    <file-put>
        <delete-if-exist />
        <encoding>value</encoding>
        <filename>value</filename>
        <permission>value</permission>
        <file-contents>file</file-contents>
    </file-put>
</rpc>
```

The following tag elements are used with the `file-put` command. These tags can be placed in any order with the exception of `file-contents`. The `file-contents` tag element must be the last tag in list.

- `delete-if-exist`—(Optional) If included, any existing file is overwritten. If the tag is omitted, an error is returned if an existing file is encountered.
- `encoding`—(Mandatory) Specifies the type of encoding used. You can use `ASCII` or `base64` encoding.
- `filename`—(Mandatory) Within this tag, you include the full or relative path and filename of the file to create. When you use a relative path, the specified path is relative to the user's home directory. If the specified directory does not exist, the system returns a "directory does not exist" error.
- `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.
- `file-contents`—(Mandatory) The ASCII or base64 encoded file contents to export. This must be the last tag in the list.

XSLT Syntax

The following sample script executes a Junos XML API request and exports the results to a file on a remote device and a file on the local device. The script takes three arguments: the IP address or hostname of the remote device, the filename, and the file encoding. The `arguments` variable is declared at the global level of the script so that the argument names and descriptions are visible in the command-line interface (CLI).

The script invokes the Junos XML API `<get-software-information>` request on the local device and stores the result in the `result` variable. The script declares the `fileput` variable, which contains the *remote procedure call* (RPC) for the `file-put` operation. The command-line arguments define the values for the `filename` and `encoding` tag elements. If the mandatory argument `myhost` is missing, the script issues an error and halts execution. Otherwise, the script prompts for the username and password that will be used to connect to the remote device.

If connection to the remote device is successful, the script executes the RPC within the context of the connection handle. The output of the `file-put` operation, which is the result of the `jcs:execute()` function, is stored in the `out` variable. If the operation encounters an error, the script prints the error to the CLI. If the `file-put` operation is successful, the contents specified by the `file-contents` tag element are exported to the specified file on the remote device. The connection to the remote host is then closed. The script also exports the contents to an identical file on the local device.

The sample script includes the optional tag elements `permission` and `delete-if-exist` for the `file-put` operation. By including the `delete-if-exist` tag, the script overwrites any existing file of the same name on the remote and local hosts. In this example, the `permission` tag is set to `0644`.

```
<?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:import href="../import/junos.xsl"/>

  <xsl:variable name="arguments">
    <argument>
      <name>myhost</name>
      <description>IP address or hostname of the remote host</description>
    </argument>
    <argument>
      <name>filename</name>
      <description>name of destination file</description>
    </argument>
    <argument>
      <name>encoding</name>
      <description>ascii or base64</description>
    </argument>
  </xsl:variable>

  <xsl:param name="myhost"/>
  <xsl:param name="filename"/>
```

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

<xsl:template match="/">
  <op-script-results>

    <xsl:variable name="rpc">
      <get-software-information/>
    </xsl:variable>
    <xsl:variable name="result" select="jcs:invoke($rpc)"/>

    <xsl:variable name="fileput">
      <file-put>
        <filename>
          <xsl:value-of select="$filename"/>
        </filename>
        <encoding>
          <xsl:value-of select="$encoding"/>
        </encoding>
        <permission>0644</permission>
        <delete-if-exist/>
        <file-contents>
          <xsl:value-of select="$result"/>
        </file-contents>
      </file-put>
    </xsl:variable>

    <xsl:choose>
      <xsl:when test="$myhost = ''">
        <xnm:error>
          <message>missing mandatory argument 'myhost'</message>
        </xnm:error>
      </xsl:when>
      <xsl:otherwise>
        <xsl:variable name="username" select="jcs:get-input('Enter username: ')"/>
        <xsl:variable name="pw" select="jcs:get-secret('Enter password: ')"/>
        <xsl:variable name="connect" select="jcs:open($myhost, $username, $pw)"/>
        <xsl:choose>
          <xsl:when test="$connect">
            <output>Connected to host.  Exporting file... </output>
            <xsl:variable name="out" select="jcs:execute($connect, $fileput)"/>
            <xsl:choose>
              <xsl:when test="$out//xnm:error">
                <xsl:copy-of select="($out//xnm:error)"/>
```

```
                </xsl:when>
                <xsl:otherwise>
                  <output>
                    <xsl:value-of select="$out"/>
                  </output>
                </xsl:otherwise>
              </xsl:choose>
              <xsl:value-of select="jcs:close($connect)"/>
            </xsl:when>
            <xsl:otherwise>
              <output>No connection to host.</output>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:otherwise>
      </xsl:choose>

      <xsl:variable name="local-out" select="jcs:invoke($fileput)"/>
      <output>
        <xsl:value-of select="concat('Saving file on local host\n', $local-out)"/>
      </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> "myhost";
          <description> "IP address or hostname of the remote host";
     }
     <argument> {
          <name> "filename";
          <description> "name of destination file";
     }
     <argument> {
```

```
        <name> "encoding";
        <description> "ascii or base64";
    }
}

param $myhost;
param $filename;
param $encoding;

match / {
    <op-script-results> {

        var $rpc = <get-software-information>;
        var $result = jcs:invoke($rpc);

        var $fileput = {
            <file-put> {
                <filename>$filename;
                <encoding>$encoding;
                <permission>'0644';
                <delete-if-exist>;
                <file-contents>$result;
            }
        }

        if ($myhost = '') {
            <xnm:error> {
                <message> "missing mandatory argument 'myhost'";
            }
        }
        else {
            var $username = jcs:get-input("Enter username: ");
            var $pw = jcs:get-secret("Enter password: ");
            var $connect = jcs:open($myhost, $username, $pw);

            if ($connect) {
                <output> "Connected to host.  Exporting file... \n";
                var $out = jcs:execute($connect, $fileput);
                if ($out//xnm:error) {
                    copy-of ($out//xnm:error);
                }
                else {
                    <output> $out;
```

```
                    }
                    expr jcs:close($connect);

               }
               else {
                    <output> "No connection to host.";
               }

          }
          var $local-out = jcs:invoke($fileput);
          <output> "Saving file on local host\n" _ $local-out;
     }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the XSLT or SLAX script into a text file, name the file **export.xsl** or **export.slax** as appropriate, and copy it to the **/var/db/scripts/op/** directory on the device.

2. In configuration mode, include the `file` statement at the `[edit system scripts op]` hierarchy level and **export.xsl** or **export.slax** as appropriate.

```
[edit system scripts op]
user@host# set file export.(slax | xsl)
```

3. Issue the `commit and-quit` command.

```
[edit]
user@host# commit and-quit
```

4. Execute the op script by issuing the `op export` operational mode command and include any necessary arguments.

Verification

IN THIS SECTION

Verifying the Op Script Arguments

Purpose

Verify that the argument names and descriptions show up in the CLI.

Action

Issue the `op exort ?` operational mode command. The CLI lists the possible completions for the script arguments based on the definitions within the global `arguments` variable in the script.

```
user@host> op export ?
Possible completions:
  <[Enter]>            Execute this command
  <name>               Argument name
  detail               Display detailed output
  encoding             ascii or base64
  filename             name of destination file
  myhost               IP address or hostname of the remote host
  |                    Pipe through a command
```

Verifying Op Script Execution

Purpose

Verify that the script behaves as expected.

Action

Issue the `op export myhost` *`host`* `encoding` *`encoding`* `filename` *`file`* operational mode command, and include the appropriate username and password when prompted. If script execution is successful, the result of the `<get-software-information>` RPC request is written to the file on the remote device and also on the local device. For example:

```
root@host> op export myhost router1 encoding ascii filename /var/log/host-version.txt
Enter username: root
Enter password:
Connected to host.  Exporting file...

/var/log/host-version.txt
Saving file on local host

/var/log/host-version.txt
```

If you fail to supply the IP address or hostname of the remote device in the command-line arguments, the script issues an error and halts execution.

```
root@host> op export
error: missing mandatory argument 'myhost'
```

If you omit the `delete-if-exist` child tag of the `file-put` operation, and the specified file already exists, the script reports an error.

```
root@host> op export myhost router1 encoding ascii filename /var/log/host-version.txt
Enter username: root
Enter password:
Connected to host.  Exporting file...

Destination file exists
Saving file on local host
```

```
Destination file exists
```

If you execute the script and include a directory path that does not exist on either the remote or the local host, the script reports an error.

```
root@host> op export myhost router1 encoding ascii filename /var/test/host-version.txt
Enter username: root
Enter password:
Connected to host.  Exporting file...

Destination directory does not exist: /var/test
Saving file on local host

Destination directory does not exist: /var/test
```

RELATED DOCUMENTATION

Declare and Use Command-Line Arguments in Op Scripts | 833
Example: Import Files Using an Op Script | 917

Example: Import Files Using an Op Script

IN THIS SECTION

The *op script* in this example uses the Junos XML protocol `file-get` operation to read the contents of a file from a remote server.

Requirements

This example uses a device running Junos OS.

Overview and Op Script

The Junos XML protocol `file-get` operation reads the contents of a file. The basic syntax for using the `file-get` command is as follows:

```
<rpc>
    <file-get>
        <filename>value</filename>
        <encoding>value</encoding>
    </file-get>
</rpc>
```

The following tag elements are used with the `file-get` command.

- `encoding`—(Mandatory) Specifies the type of encoding used. You can use `ASCII`, `base64`, or `raw` encoding.
- `filename`—(Mandatory) Within this tag, you include the full or relative path and filename of the file to import. When you use a relative path, the specified path is relative to the **/var/tmp/** directory if the `file-get` operation is executed locally. If the operation is executed remotely within the context of a connection handle, the path is relative to the user's home directory.

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).

XSLT Syntax

The following sample script connects to a remote device and reads the contents of the specified file. The script takes three arguments: the IP address or hostname of the remote device, the filename, and the file encoding. The `arguments` variable is declared at the global level of the script so that the argument names and descriptions are visible in the command-line interface (CLI).

The script declares the `fileget` variable, which contains the *remote procedure call* (RPC) for the `file-get` operation. The command-line arguments define the values for the `filename` and `encoding` tag elements. If the mandatory argument `myhost` is missing, the script issues an error and halts execution. Otherwise, the script prompts for the username and password that will be used to connect to the remote device.

If connection to the remote device is successful, the script executes the RPC within the context of the connection handle. The output of the `file-get` operation, which is the result of the `jcs:execute()` function, is stored in the `out` variable. If the operation encounters an error, the script prints the error to the CLI. If the `file-get` operation is successful, the contents of the file are stored in the `out` variable, which is printed to the CLI. The connection to the remote host is then closed.

```
<?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:import href="../import/junos.xsl"/>

  <xsl:variable name="arguments">
     <argument>
          <name>myhost</name>
          <description>IP address or hostname of the remote host</description>
     </argument>
     <argument>
          <name>filename</name>
          <description>name of file</description>
     </argument>
     <argument>
          <name>encoding</name>
          <description>ascii, base64, or raw</description>
     </argument>
  </xsl:variable>

  <xsl:param name="myhost"/>
  <xsl:param name="filename"/>
  <xsl:param name="encoding"/>

  <xsl:template match="/">
     <op-script-results>
          <xsl:variable name="fileget">
               <file-get>
                    <filename>
                         <xsl:value-of select="$filename"/>
                    </filename>
                    <encoding>
                         <xsl:value-of select="$encoding"/>
                    </encoding>
               </file-get>
          </xsl:variable>
```

```
          <xsl:choose>
               <xsl:when test="$myhost = ''">
                    <xnm:error>
                         <message>missing mandatory argument 'myhost'</message>
                    </xnm:error>
               </xsl:when>
               <xsl:otherwise>
                    <xsl:variable name="username" select="jcs:get-input('Enter username: ')"/>
                    <xsl:variable name="pw" select="jcs:get-secret('Enter password: ')"/>
                    <xsl:variable name="connect" select="jcs:open($myhost, $username, $pw)"/>
                    <xsl:choose>
                         <xsl:when test="$connect">
                              <output>Connected to host.  Reading file...
                              </output>
                              <xsl:variable name="out" select="jcs:execute($connect, $fileget)"/>
                              <xsl:choose>
                                   <xsl:when test="$out//xnm:error">
                                        <xsl:copy-of select="$out//xnm:error"/>
                                   </xsl:when>
                                   <xsl:otherwise>
                                        <output>
                                             <xsl:value-of select="concat('File contents: ',
$out)"/>
                                        </output>
                                   </xsl:otherwise>
                              </xsl:choose>
                              <xsl:value-of select="jcs:close($connect)"/>
                         </xsl:when>
                         <xsl:otherwise>
                              <output>No connection to host.</output>
                         </xsl:otherwise>
                    </xsl:choose>
               </xsl:otherwise>
          </xsl:choose>
     </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> "myhost";
          <description> "IP address or hostname of the remote host";
     }
     <argument> {
          <name> "filename";
          <description> "name of file";
     }
     <argument> {
          <name> "encoding";
          <description> "ascii, base64, or raw";
     }
}

param $myhost;
param $filename;
param $encoding;

match / {
     <op-script-results> {
          var $fileget = {
               <file-get> {
                    <filename>$filename;
                    <encoding>$encoding;
               }
          }

          if ($myhost = '') {
               <xnm:error> {
                    <message> "missing mandatory argument 'myhost'";
               }
          }
          else {
               var $username = jcs:get-input("Enter username: ");
               var $pw = jcs:get-secret("Enter password: ");
               var $connect = jcs:open($myhost, $username, $pw);

               if ($connect) {
```

```
                    <output> "Connected to host.  Reading file... \n";
                    var $out = jcs:execute($connect, $fileget);
                    if ($out//xnm:error) {
                         copy-of $out//xnm:error;
                    }
                    else {
                         <output> "File contents: " _ $out;
                    }
                    expr jcs:close($connect);
               }
               else {
                    <output> "No connection to host.";
               }
          }
     }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the XSLT or SLAX script into a text file, name the file **import.xsl** or **import.slax** as appropriate, and copy it to the **/var/db/scripts/op/** directory on the device.

2. In configuration mode, include the `file` statement at the `[edit system scripts op]` hierarchy level and **import.xsl** or **import.slax** as appropriate.

   ```
   [edit system scripts op]
   user@host# set file import.(slax | xsl)
   ```

3. Issue the `commit and-quit` command to commit the configuration and to return to operational mode.

```
[edit]
user@host# commit and-quit
```

4. Execute the op script by issuing the `op import` operational mode command and include any necessary arguments.

Verification

IN THIS SECTION

Verifying the Script Arguments

Purpose

Verify that the argument names and descriptions show up in the CLI.

Action

Issue the `op import ?` operational mode command. The CLI lists the possible completions for the script arguments based on the definitions within the global `arguments` variable in the script.

```
user@host> op import ?
Possible completions:
  <[Enter]>            Execute this command
  <name>               Argument name
  detail               Display detailed output
  encoding             ascii, base64, or raw
  filename             name of file
  myhost               IP address or hostname of the remote host
  |                    Pipe through a command
```

Verifying Op Script Execution

Purpose

Verify that the script behaves as expected.

Action

Issue the `op import myhost` *`host`* `encoding` *`encoding`* `filename` *`file`* operational mode command, and include the appropriate username and password when prompted. If script execution is successful, the contents of the requested file are displayed. For example:

```
root@host> op import myhost router1 encoding ascii filename /var/db/scripts/op/test.slax
Enter username: root
Enter password:
Connected to host.  Reading file...
File contents:

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";
...
```

If you fail to supply the IP address or hostname of the remote device in the command-line arguments, the script issues an error and halts execution.

```
root@host> op import
error: missing mandatory argument 'myhost'
```

Also, if the specified path or file does not exist, the script issues an error.

```
root@host> op import myhost router1 encoding ascii filename /var/db/scripts/op/test1.slax
Enter username: root
Enter password:
Connected to host.  Reading file...
File contents:

Failed to open file (/var/db/scripts/op/test1.slax): No such file or directory
```

RELATED DOCUMENTATION

Example: Search Files Using an Op Script

IN THIS SECTION

This sample script searches a file on a device running Junos OS for lines matching a given regular expression. The example uses the `jcs:grep` template in an *op script*.

Requirements

This example uses a device running Junos OS.

Overview and Op Script

The `jcs:grep` template searches an *ASCII* file for lines matching a regular expression. The template resides in the **junos.xsl** import file, which is included with the standard Junos OS installation available on all switches, routers, and security devices running Junos OS. To use the `jcs:grep` template in a script, you must import the **junos.xsl** file into the script and map the `jcs` prefix to the namespace identified by the URI http://xml.juniper.net/junos/commit-scripts/1.0 .

In this example, all values required for the `jcs:grep` template are defined as global parameters. The values for the parameters are passed into the script as command-line arguments. The following script defines two parameters, `filename` and `pattern`, which store the values of the input file path and the regular expression. If you omit either argument when you execute the script, the script generates an error and halts execution. Otherwise, the script calls the `jcs:grep` template and passes in the supplied arguments.

If the regular expression contains a syntax error, the `jcs:grep` template generates an `error: regex error` message for each line in the file. If the regular expression syntax is valid, the template parses the input file. For each match, the template adds a `<match>` element, which contains `<input>` and `<output>` child tags, to the result tree. The template writes the matching string to the `<output>` child element and writes the corresponding matching line to the `<input>` child element:

```
<match> {
     <input>
     <output>
}
```

In the SLAX script, the `:=` operator copies the results of the `jcs:grep` template call to a temporary variable and runs the `node-set` function on that 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 XSLT script explicitly calls out the equivalent steps. The script then loops through all resulting input elements and prints each match.

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:import href="../import/junos.xsl"/>

     <xsl:variable name="arguments">
          <argument>
               <name>filename</name>
               <description>name of file in which to search for the specified pattern
               </description>
          </argument>
          <argument>
               <name>pattern</name>
               <description>regular expression</description>
          </argument>
     </xsl:variable>

     <xsl:param name="filename"/>
     <xsl:param name="pattern"/>

     <xsl:template match="/">

          <op-script-results>
               <xsl:choose>
```

```
                    <xsl:when test="$filename = ''">
                         <xnm:error>
                              <message>missing mandatory argument 'filename'</message>
                         </xnm:error>
                    </xsl:when>
                    <xsl:when test="$pattern = '';">
                         <xnm:error>
                              <message>missing mandatory argument 'pattern'</message>
                         </xnm:error>
                    </xsl:when>
                    <xsl:otherwise>
                         <xsl:variable name="results-temp">
                              <xsl:call-template name="jcs:grep">
                                   <xsl:with-param name="filename" select="$filename"/>
                                   <xsl:with-param name="pattern" select="$pattern"/>
                              </xsl:call-template>
                         </xsl:variable>
                         <xsl:variable xmlns:ext="http://xmlsoft.org/XSLT/namespace"
                              name="results" select="ext:node-set($results-temp)"/>
                         <output>
                              <xsl:value-of select="concat('Search for ', $pattern, ' in ',
$filename)"/>
                         </output>
                         <xsl:for-each select="$results//input">
                              <output>
                                   <xsl:value-of select="."/>
                              </output>
                         </xsl:for-each>
                    </xsl:otherwise>
               </xsl:choose>
          </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> "filename";
         <description> "name of file in which to search for the specified pattern";
     }
     <argument> {
         <name> "pattern";
         <description> "regular expression";
     }
}

param $filename;
param $pattern;

match / {
     <op-script-results> {

          if ($filename = '') {
               <xnm:error> {
                    <message> "missing mandatory argument 'filename'";
               }
          }
          else if ($pattern = '') {
               <xnm:error> {
                    <message> "missing mandatory argument 'pattern'";
               }
          }
          else {
               var $results := { call jcs:grep($filename, $pattern);  }

               <output> "Search for " _ $pattern _ " in " _ $filename;
               for-each ($results//input) {
                    <output> .;
               }
          }
     }
}
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download, enable, and run the script:

1. Copy the XSLT or SLAX script into a text file, name the file **grep.xsl** or **grep.slax** as appropriate, and download it to the **/var/db/scripts/op/** directory on the device.

2. In configuration mode, include the `file` statement at the `[edit system scripts op]` hierarchy level and **grep.xsl** or **grep.slax** as appropriate.

```
[edit system scripts op]
user@host# set file grep.(slax | xsl)
```

3. Issue the `commit and-quit` command to commit the configuration and to return to operational mode.

```
[edit]
user@host# commit and-quit
```

4. Execute the op script by issuing the `op grep filename` *`filename`* `pattern` *`pattern`* operational mode command.

Verification

IN THIS SECTION

Verifying the Script Arguments

Purpose

Verify that the argument names and descriptions appear in the command-line interface (CLI) help.

Action

Issue the `op grep ?` operational mode command. The CLI lists the possible completions for the script arguments based on the definitions within the global variable `arguments` in the script.

```
user@host> op grep
Possible completions:
  <[Enter]>            Execute this command
  <name>               Argument name
  detail               Display detailed output
  filename             name of file in which to search for the specified pattern
  pattern              regular expression
  |                    Pipe through a command
```

Verifying Op Script Execution

Purpose

Verify that the script behaves as expected.

Action

If you issue the `op grep` command, but you fail to supply either the filename or the regex pattern, the script issues an error message and halts execution. For example:

```
user@host> op grep filename /var/log/messages
error: missing mandatory argument 'pattern'

user@host> op grep pattern SNMP_TRAP_LINK_DOWN
error: missing mandatory argument 'filename'
```

When you issue the `op grep filename` *`filename`* `pattern` *`pattern`* command, the script lists all lines from the input file that match the regular expression.

```
user@host> op grep filename /var/log/messages pattern SNMP_TRAP_LINK_DOWN
Search for SNMP_TRAP_LINK_DOWN in /var/log/messages
Feb 24 09:04:00  host mib2d[1325]: SNMP_TRAP_LINK_DOWN: ifIndex 543, ifAdminStatus down(2),
ifOperStatus down(2), ifName
lt-0/1/0.9
Feb 24 09:04:00  host mib2d[1325]: SNMP_TRAP_LINK_DOWN: ifIndex 542, ifAdminStatus down(2),
ifOperStatus down(2), ifName
lt-0/1/0.10
```

RELATED DOCUMENTATION

SLAX Templates Overview | 97
jcs:grep Template | 440
regex() Function (SLAX and XSLT) | 409

CHAPTER 32

Provision Services Using Service Template Automation

IN THIS CHAPTER

Service Template Automation Overview

Starting in Junos OS Release 12.3, you can use service template automation (STA) to provision services such as VPLS VLAN, Layer 2 and Layer 3 VPNs, and IPsec across similar platforms running Junos OS. Service template automation uses the **service-builder.slax** op script to transform a user-defined service template definition into a uniform API, which you can then use to configure and provision services on similar platforms running Junos OS. This enables you to create a service template on one device, generalize the parameters, and then quickly and uniformly provision that service on other devices. This decreases the time required to configure the same service on multiple devices, and reduces configuration errors associated with manually configuring each device.

The following process outlines how to use service template automation to provision services:

1. Create a service template definition.

2. Execute the **service-builder.slax** script and define service-specific instance parameters.

3. Generate the service interface, which automatically builds the required interface (API) from the template.

4. Enable the service interface on each device where the service is required.

5. Provision systems by invoking the service interface using NETCONF and supplying the service parameter values.

You create a new service template by configuring the hierarchies for the actual service to be provisioned on a device running Junos OS. Service template hierarchies are configured at the `[edit groups]` hierarchy level. When creating the service template:

- Do not include `apply-groups` or `apply-macro` statements.

- Do not include any statements that are supported on the current device that are not also supported on the devices where the service will be provisioned (for example dual Routing Engine versus single Routing Engine).

- Commit the configuration. The service template group configuration is read from the committed configuration.

Once you create the basic service template definition, you invoke the **service-builder.slax** op script. The script reads the service template information from the committed configuration and uses an interactive interface to help you build and generate the service API. You have the option to parameterize every variable in the service template or only selected variables. For each selected variable, you create a generic service template parameter. The **service-builder.slax** script guides you through the creation and configuration of each parameter.

After you define the service template parameters, you generate the service interface. This creates a platform-specific service op script. If the `load-scripts-from-flash` statement is configured, the generated service script is stored in the **/config/scripts/op** directory in flash memory. Otherwise, the generated script is stored in the **/var/db/scripts/op** directory on the hard disk.

To enable the service interface on a device, you enable the generated service script in the configuration as you would any op script. You can enable the service interface on the local device using the **service-builder.slax** script or by manually updating the configuration. To enable the service interface on a similar platform, you must copy the generated service script to the corresponding directory on the new device and enable the service script in the configuration.

To provision the service on a device, invoke the service interface using NETCONF, and supply the necessary values for each parameter. Alternatively, you can invoke the service interface in the CLI by executing the service script and supplying the necessary values for each parameter as command-line arguments to the script. You can direct the service script to create a new service configuration, or update or delete an existing service configuration. The service script makes the changes to the candidate configuration and then commits the configuration. The service script does not support the context-sensitive help and auto-completion features available in the Junos OS CLI.

RELATED DOCUMENTATION

Example: Configure Service Template Automation | **934**

Example: Configure Service Template Automation

IN THIS SECTION

This example shows how to use service template automation to provision services across similar platforms running Junos OS.

Requirements

- Two MX Series devices running Junos OS Release 12.3 or later.

Overview

This example uses service template automation to provision services on an MX Series router. To use the service template automation **service-builder.slax** script, you must first copy the script to the **/var/db/scripts/op** or **/config/scripts/op** directory and enable the script on the device.

The following process outlines how to use service template automation to provision services:

1. Create a service template definition.
2. Execute the **service-builder.slax** script, and define service-specific instance parameters.
3. Generate the service interface.
4. Enable the service interface on each device where the service is required.
5. Provision systems by invoking the service interface using NETCONF and supplying the service parameter values.

This example creates a new VPN service interface on an MX Series device running Junos OS Release 12.3 and provisions the service on a second MX Series device running Junos OS Release 12.3. You configure service template definitions under the `[edit groups]` hierarchy level. For this example, the service name is `vpn-service`, and the template group name is `vpn-service-template-group`. The `load merge` `terminal` configuration mode command loads the service template configuration hierarchies into the candidate configuration, which is then committed.

Once you create the initial service template, you execute the **service-builder.slax** script. The script prompts for the service name and the template group name, and then reads the service template configuration from the committed configuration.

The **service-builder.slax** script interface consists of two menus: `Main Menu` and `Hierarchies Menu`. Within the `Main Menu`, you can review the variables defined in the service template configuration, or you can build or enable the service API. The `Build Service API` menu option displays the `Hierarchies Menu`, which steps you through the parameterization of the variables. The default is to parameterize every variable, or you can choose to parameterize selected variables. If you must exit the **service-builder.slax** script while building the service API, you must finish configuring all the parameters for the current hierarchy in order to save that hierarchy configuration when you exit using the `Quit` option. Then you can finish configuring any incomplete hierarchies at a later time. This example parameterizes two variables: the interface name and the interface description. After the parameters are specified, the service builder script generates the service script.

The `Enable Service API` menu option enables the service script on the local device. To enable the service script on the second MX Series device, the generated service script is copied to the **/var/db/scripts/op** directory on the second device, and the script is enabled in the configuration. If the `load-scripts-from-flash` statement is configured, the script must be copied to the corresponding directory on the flash drive instead.

NETCONF is used to provision the service on the remote MX Series device. The NETCONF remote procedure call (RPC) action depends on whether the service is a new service or an existing service. Supported actions include `create`, `update`, and `delete`. This example creates a new service. If the given service is already provisioned on the device and you are updating or deleting the service parameters, you can alter the RPC to perform these actions.

Configuration

IN THIS SECTION

Storing and Enabling the Service Builder Script

Step-by-Step Procedure

The Junos OS installation includes the **service-builder.slax** script, which is stored in the **/usr/libexec/scripts/op/** directory on the device. To use the **service-builder.slax** script, you must first copy it to the op scripts directory and enable it in the configuration. Only users in the Junos OS superuser login class can access and edit files in these directories.

1. Copy the **service-builder.slax** script to the **/var/db/scripts/op** directory on the hard disk or the **/config/scripts/op** directory on the flash drive.

```
user@host> file copy /usr/libexec/scripts/op/service-builder.slax /var/db/scripts/op
```

2. Verify that the script is in the correct directory by using the `file list` operational mode command.

```
user@host> file list /var/db/scripts/op
/var/db/scripts/op:
service-builder.slax*
```

3. Enable the script in the configuration.

```
[edit]
user@host# set system scripts op file service-builder.slax
```

4. If you store scripts in and load them from flash, configure the `load-scripts-from-flash` statement, if it is not already configured.

```
[edit]
user@host# set system scripts load-scripts-from-flash
```

5. Commit the configuration.

```
[edit]
user@host# commit
```

Configuring the Service Template Definition

Step-by-Step Procedure

To create a new service template on a device running Junos OS:

1. Select a service name.

   This example uses `vpn-service`.

2. In configuration mode, create a new group, which will contain the hierarchies for the actual service to be provisioned.

```
[edit]
user@host# set groups vpn-service-template-group
```

3. Configure the hierarchies for the service.

   For this example, the pre-constructed configuration hierarchies are loaded into the candidate configuration using the `load merge terminal` command.

```
[edit]
user@host# load merge terminal
groups {
    vpn-service-template-group {
        interfaces {
            ge-2/2/6 {
                description "connected to customer3-site-1";
                unit 0 {
                    family bridge {
                        interface-mode access;
                        vlan-id 300;
                    }
                }
            }
        }
        protocols {
            rstp {
                interface ge-2/3/0;
            }
            mvrp {
                interface ge-2/3/0;
```

```
            }
        }
        bridge-domains {
            bd {
                vlan-id-list 100;
            }
        }
    }
}
[Ctrl+D]
```

4. Verify that the configuration syntax is correct.

```
[edit]
user@host# commit check
configuration check succeeds
```

5. Commit the configuration.

```
[edit]
user@host# commit
```

Configuring and Generating the Service Interface

Step-by-Step Procedure

To configure and generate the service interface:

1. In operational mode, execute the **service-builder.slax** script, which starts an interactive Service Builder session.

```
user@host> op service-builder
Welcome to Service Builder Script: (v1.0)
-
Enter the service name :
```

2. Enter the service name that was defined in "Configuring the Service Template Definition" on page 937.

```
Enter the service name : vpn-service
```

3. Enter the group name under which the service hierarchies are configured.

   This example uses the group name `vpn-service-template-group`. The script reads the configuration specified in the `vpn-service-template-group` hierarchy and then displays the main menu.

```
Enter the group name : vpn-service-template-group
.. reading [edit group vpn-service-template-group] ..

[Op Script Builder - Main Menu]
-----------------------------------
 1. Show Variables
 2. Build Service API
 3. Enable Service API
 Q. Quit
-----------------------------------
Enter Selection:>
```

4. (Optional) To review the service template variables that you can parameterize, select the `Show Variables` option.

   The script translates the template definition in the candidate configuration into a general parameter list grouped by hierarchy level.

```
[Op Script Builder - Main Menu]
-----------------------------------
 1. Show Variables
 2. Build Service API
 3. Enable Service API
 Q. Quit
-----------------------------------
Enter Selection:> 1
 List of variables under each hierarchy to parameterize:
-
 1. [ edit groups vpn-service-template-group interfaces ]
```

```
  1.1. interface/name
  1.2. interface/description
  1.3. interface/unit/name
  1.4. interface/unit/family/bridge/interface-mode
  1.5. interface/unit/family/bridge/vlan-id
-
 2. [ edit groups vpn-service-template-group protocols ]
  2.1. rstp/interface/name
  2.2. mvrp/interface/name
-
 3. [ edit groups vpn-service-template-group bridge-domains ]
  3.1. domain/name
  3.2. domain/vlan-id-list
-
```

5. To build the Service API, select the `Build Service API` option.

```
[Op Script Builder - Main Menu]
-----------------------------------
 1. Show Variables
 2. Build Service API
 3. Enable Service API
 Q. Quit
-----------------------------------
Enter Selection:> 2
```

6. From the `Hierarchies Menu`, enter the menu selections for the hierarchies that have variables you want to parameterize, or press Enter to select all hierarchies.

```
[Op Script Builder - Hierarchies Menu]
-----------------------------------
 1. interfaces
 2. protocols
 3. bridge-domains
 Q. Quit
-----------------------------------
Please enter multiple selections separated by a comma (,) only.
-----------------------------------
```

```
Enter Selection:> [default:all] 1
```

7. From the variables list, enter the menu selections for the variables you want to parameterize for the service interface, or press Enter to parameterize all variables within that hierarchy.

```
List of variables to parameterize: ...
[ edit groups vpn-service-template-group interfaces ]
-----------------------------------
 1. interfaces/interface/name
 2. interfaces/interface/description
 3. interfaces/interface/unit/name
 4. interfaces/interface/unit/family/bridge/interface-mode
 5. interfaces/interface/unit/family/bridge/vlan-id
 Q. Quit
Please enter multiple selections separated by a comma (,) only.
-----------------------------------
Enter Selection:> [default:all] 1,2
```

8. Configure the selected parameters.

   The system prompts for the required information. This example configures the interface name parameter as `ifname` and the interface description parameter as `ifdesc`.

```
Enter parameter name for: 1.interfaces/interface/name
****************************
[ edit groups vpn-service-template-group interfaces ]
Name for this parameter? ifname
Do you want to revise 'ifname'? (yes/no)[no]: no
Enter parameter name for: 2.interfaces/interface/description
****************************
[ edit groups vpn-service-template-group interfaces ]
Name for this parameter? ifdesc
Do you want to revise 'ifdesc'? (yes/no)[no]: no
```

9. Configure the selected parameters at each hierarchy level.

   The script iterates over each selected hierarchy and the specified parameters. If you must exit the **service-builder.slax** script while building the service API, you must finish configuring all the parameters for the current hierarchy in order to save that hierarchy configuration when you exit using the `Quit` option.

10. Generate the service interface, which creates the service script.

    Once all parameters are configured, the script automatically prompts you to generate the service interface. Press Enter or type yes to generate the service interface.

```
Do you want to commit the previously selected options to create vpn-service script?
(yes/no)[yes]: yes
Created service script: /var/db/scripts/op/vpn-service.slax
```

Verifying the Service Interface

Purpose

Verify the creation of the service script. If the `load-scripts-from-flash` statement is configured, the generated file is stored in flash memory. Otherwise, the generated file is stored on the hard disk.

Action

Issue the `file list` operational mode command. For this example, the **vpn-service.slax** script should be present in the **/var/db/scripts/op** directory. The **service-builder.slax** script also generates the **utility.slax** script in the **/var/db/scripts/op** directory and the **vpn-service-builder-info.xml** file in the **/var/db/scripts/lib** directory. These files are used by the **service-builder.slax** script and should not be deleted.

```
user@host> file list /var/db/scripts/op
/var/db/scripts/op:
service-builder.slax
utility.slax
vpn-service.slax

user@host> file list /var/db/scripts/lib
/var/db/scripts/lib:
vpn-service-builder-info.xml
```

Enabling the Service Interface

Step-by-Step Procedure

To enable the service interface on a remote device:

1. Copy the generated service script to the device where you are provisioning the new service.

   If the `load-scripts-from-flash` statement is not configured, copy the service script to the **/var/db/scripts/op** directory on the second device. Otherwise, the script must be copied to the corresponding directory on the flash drive instead.

2. Enable the op script in the configuration.

```
[edit]
user@host2# set system scripts op file vpn-service.slax
```

3. Commit the configuration.

```
[edit]
user@host2# commit
commit complete
```

4. In operational mode, verify that the script is enabled and that the service parameters display as arguments for the script.

```
user@host2> op vpn-service ?
Possible completions:
 <[Enter]>   Execute this command
 <name>      Argument name
 action      Please enter either create/delete/update
 detail      Display detailed output
 ifdesc      Text description of interface
 ifname      Name of interface
 service-id  Service Name
 |           Pipe through a command
```

Provisioning the Service Using NETCONF

Step-by-Step Procedure

To provision the service:

1. If it is not already configured, configure NETCONF service over SSH on any devices where you are provisioning the new service.

```
[edit]
user@host2# set system services netconf ssh
user@host2# commit
```

2. From a configuration management server, establish a NETCONF session with the device where you are provisioning the service.

```
%ssh -p 830 -s user@host2 netconf
user@host2's password:

<!-- user user, class super-user -->
<hello>
  <capabilities>
   <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
   <capability>
      urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
   </capability>
   <capability>
      urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
   </capability>
   <capability>
      urn:ietf:params:xml:ns:netconf:capability:validate:1.0
   </capability>
   <capability>
      urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
    </capability>
   <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
   <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>28898</session-id>
</hello>
]]>]]>
```

3. If you are provisioning a new service on the device, enter a remote procedure call (RPC) that calls the service op script using the `create` action, and include values for all parameters that require configuring.

   The value for the `service-id` parameter should be identical to the service name.

```
<rpc>
  <op-script>
    <script>vpn-service</script>
    <action>create</action>
    <service-id>vpn-service</service-id>
    <ifname>ge-2/0/5,ge-2/0/6</ifname>
    <ifdesc>connected to customer1-site-1,connected to customer3-site-2</ifdesc>
  </op-script>
</rpc>
```

Updating or Deleting Services Using NETCONF

Step-by-Step Procedure

To update or delete an existing service:

1. If it is not already configured, configure NETCONF service over SSH on any devices where you are updating or deleting the service.

```
[edit]
user@host2# set system services netconf ssh
user@host2# commit
```

2. From a configuration management server, establish a NETCONF session with the device where you are provisioning the service.

```
%ssh -p 830 -s user@host2 netconf
user@host2's password:

<!-- user user, class super-user -->
<hello>
  <capabilities>
   <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
   <capability>
      urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
```

```
    </capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
    </capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:validate:1.0
    </capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
     </capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>28898</session-id>
</hello>
]]>]]>
```

3. If the given service is already provisioned on the device and you are updating the service, enter an RPC that calls the service op script using the `update` action, and include values for all parameters that require updating.

```
<rpc>
  <op-script>
    <script>vpn-service</script>
    <action>update</action>
    <service-id>vpn-service</service-id>
    <ifname>ge-2/0/5</ifname>
    <ifdesc>connected to customer1-site-2</ifdesc>
  </op-script>
</rpc>
```

4. If the given service is already provisioned on the device and you are deleting some or all of the service parameters, enter an RPC that calls the service op script using the `delete` action, and include any parameters that need to be deleted.

```
<rpc>
  <op-script>
    <script>vpn-service</script>
    <action>update</action>
    <service-id>vpn-service</service-id>
    <ifname>ge-2/0/6</ifname>
```

```
  </op-script>
</rpc>
```

Verification

IN THIS SECTION

Confirm that the configuration is updated.

Verifying the Commit

Purpose

Verify that the commit is successful.

Action

Issue the `show system commit` operational mode command to view the recent commits. The most recent commit entry shows that a commit was made through the NETCONF server by `user`.

```
user@host2> show system commit

0   2012-05-21 12:15:08 PDT by user via junoscript
1   2012-05-18 09:47:40 PDT by user via other
...
```

Verifying the Service Configuration

Purpose

Verify that the service configuration is present in the active configuration.

Action

Issue the `show configuration | compare rollback` *num* operational mode command to view configuration changes.

```
user@host2> show configuration | compare rollback 1

[edit interfaces]
+   ge-2/0/5 {
+       description "connected to customer1-site-1";
+   }
+   ge-2/0/6 {
+       description "connected to customer3-site-2";
+   }
```

Meaning

A comparison of the current configuration with the previous configuration shows that the interfaces and interface descriptions were added to the configuration.

Troubleshooting

IN THIS SECTION

Troubleshooting a Failed Commit

Problem

You see the following message when creating, updating, or deleting a service on a device through a NETCONF session:

```
<output>
configuration database modified
</output>
```

The configuration has previously uncommitted changes, and the service script cannot commit the service configuration changes.

Solution

Commit the previous changes or roll back the configuration as appropriate, and then resubmit the service configuration changes.

Troubleshooting a Failed Attempt to Delete Service Parameters

Problem

You see the following message when deleting a service parameter on a device through NETCONF:

```
<xnm:error xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
  <source-daemon>
    op-script
  </source-daemon>
  <message>
    xsl:attribute: Cannot add attributes to an element if children have been already added to
the element.
  </message>
</xnm:error>
```

Solution

The RPC might include both the parameter and a child element. Remove the child element from the RPC.

RELATED DOCUMENTATION

Service Template Automation Overview | 932

CHAPTER 33

Troubleshoot Op Scripts

IN THIS CHAPTER

Trace Op Script Processing on Devices Running Junos OS

IN THIS SECTION

Op script tracing operations track 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, which include errors, warnings, progress messages, and script processing events, in the **/var/log/op-script.log** file on the device. 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.

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. Commit the configuration.

```
[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" on page 952. 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 52 on page 952 summarizes useful filtering commands that display selected portions of the `op-script.log` file.

Table 52: 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`

Table 52: Op Script Tracing Operational Mode Commands *(Continued)*

Task	Command
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 Local Op Scripts

You cannot change the directory (**/var/log**) to which trace files are written. However, you can customize other trace file settings for local op scripts 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;
```

NOTE: You can only customize tracing operations for local op scripts. For remote op scripts (op scripts executed using the `op url` command), Junos OS always logs the default trace events in the default op script log file.

These statements are described in the following sections:

Configuring the Op Script Log Filename

By default, the name of the file that records trace output for all op scripts is **op-script.log**. You can record the trace output for local op scripts in a different file by including the `file` statement at the `[edit system scripts op traceoptions]` hierarchy level:

```
[edit system scripts op traceoptions]
file filename;
```

NOTE: The `file` statement does not affect the trace output location for remote op scripts, which always use the default log file **op-script.log**.

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, the traceoptions `events` flag is turned on, regardless of the configuration settings, and only important events are logged. This includes errors, warnings, progress messages, and script processing events. You can configure the trace operations to be logged for local op scripts 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;
```

NOTE: The `flag` statement does not affect the trace output for remote op scripts, which only log default trace events.

Table 53 on page 956 describes the meaning of the op script tracing flags.

Table 53: Op Script Tracing Flags

Flag	Description	Default Setting
`all`	Trace all operations.	Off
`events`	Trace important events, including errors, warnings, progress messages, and script processing 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

Trace Script Processing on Devices Running Junos OS Evolved

IN THIS SECTION

When you execute an interactive script, the script can generate output, including warnings and errors, in the CLI or RPC reply. When the system triggers non-interactive scripts, for example, when an event policy triggers an event script, the script does not direct output to the terminal. In either case, you might need more information about the execution of the script. Junos OS Evolved captures trace data for all applications by default. You can view the collected traces for additional script processing information, including the memory and CPU usage, script arguments, script execution, and warnings and errors.

Junos OS Evolved collects trace data from all applications on all nodes on the Routing Engine. Whereas Junos OS logs the trace data for each type of script in separate log files, Junos OS Evolved stores the trace data for all scripts in the same location. The trace log includes data for commit, event, op, and SNMP scripts; YANG action and translation scripts; and Juniper Extension Toolkit scripts.

How to Display Trace Data for Scripts

Junos OS Evolved stores the trace data from all nodes that is collected on the primary Routing Engine under the **/var/log/traces** directory. The `cscript` application handles scripts, and the trace data for scripts is stored under the ***node*.cscript.*sequence-number*** subdirectories.

To view trace data for scripts, issue the `show trace application cscript` operational mode command.

```
user@host> show trace application cscript
2021-05-20 09:11:42.239695672 re0:cscript:4176:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg =
"Process's current softlimit [134217728] hardlimit [-1]"
2021-05-20 09:11:42.239773157 re0:cscript:4176:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg =
"Process's limits are already set by parent process"
2021-05-20 09:11:42.239812430 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"op script processing begins"
2021-05-20 09:11:42.239855140 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: /usr/libexec/ui/cscript"
2021-05-20 09:11:42.239865140 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -mop"
2021-05-20 09:11:42.239866196 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -p"
2021-05-20 09:11:42.239867156 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: /"
2021-05-20 09:11:42.239868116 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -Q2"
2021-05-20 09:11:42.239869131 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -f"
2021-05-20 09:11:42.239882048 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: hello.py"
2021-05-20 09:11:42.239883202 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -d134217728"
```

```
2021-05-20 09:11:42.239884135 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -E"
2021-05-20 09:11:42.239885131 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg:  user admin logname admin host host tty /dev/pts/0 agent op-script current-
directory /var/home/admin pid 32212 ppid 32206"
2021-05-20 09:11:42.239886175 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -u"
2021-05-20 09:11:42.239887176 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: admin"
2021-05-20 09:11:42.239888251 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -U"
2021-05-20 09:11:42.239889287 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -i9"
2021-05-20 09:11:42.245988806 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"running op script 'hello.py'"
2021-05-20 09:11:42.246006519 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"opening op script '/var/db/scripts/op/hello.py'"
...
```

You can include the `terse` option to display just the timestamp and message.

```
user@host> show trace application cscript terse
2021-05-20 09:11:42.239695672 msg = "Process's current softlimit [134217728] hardlimit [-1]"
2021-05-20 09:11:42.239773157 msg = "Process's limits are already set by parent process"
2021-05-20 09:11:42.239812430 msg = "op script processing begins"
...
```

You can also refine the traces to display by specifying the trace time elapsed, process ID, and node. For example, the following command shows trace data for a specific process ID.

```
user@host> show trace application cscript pid 10683
2021-05-24 09:42:09.552687492 re0:cscript:10683:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg
= "Process's current softlimit [134217728] hardlimit [-1]"
2021-05-24 09:42:09.552819712 re0:cscript:10683:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg
= "Process's limits are already set by parent process"
2021-05-24 09:42:09.552897412 re0:cscript:10683:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg
= "action script processing begins"
2021-05-24 09:42:09.553025992 re0:cscript:10683:TRACE_INFO
CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1  msg = "arg: /usr/libexec/ui/cscript"
2021-05-24 09:42:09.553095062 re0:cscript:10683:TRACE_INFO
```

```
CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1  msg = "arg: -maction"
...
```

How to Modify Trace Settings for Scripts

Junos OS Evolved traces script processing by default and traces all applications at the info level for informational messages. You can configure trace settings for specific applications at the `[edit system application]` hierarchy level. For example, you can specify the trace level of the application on a given node.

To modify script tracing operations, configure the settings under the `[edit system trace application cscript]` hierarchy level. The following example configures the `cscript` application to trace script processing on node re0 at the debug level:

```
[edit]
user@host# set system trace application cscript node re0 level debug
user@host# commit
```

For more information about configuring trace settings, see *trace*.

10

PART

Event Policies and Event Scripts

CHAPTER 34

Event Policy Overview

IN THIS CHAPTER

Event Policies and Event Notifications Overview

IN THIS SECTION

Benefits of Event Policies

Event policies provide the following benefits:

- Improve network reliability and maximize network uptime by automatically responding to system events, including system log messages, SNMP traps, chassis alarms, and internal timers
- Shorten troubleshooting time and speed time to resolution for network issues by automating troubleshooting tasks
- Reduce the time required for manual system monitoring and intervention

Understanding Event Policies and Event Notifications

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 process called the *event process* (*eventd*) receives event notifications—henceforth simply called *events*—from other Junos processes.

Timely diagnosis and intervention can correct error conditions and keep the device in operation. When the eventd process receives events, it can trigger an *event policy*. Event policies are if-then-else constructs that instruct the eventd process to select specific events, correlate the events, and perform a set of actions upon receipt of the events. These actions can help you diagnose a fault as well as take corrective action. For example, the eventd process can create log files, upload device files to a given destination, issue operational mode commands, modify the configuration, 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.

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.

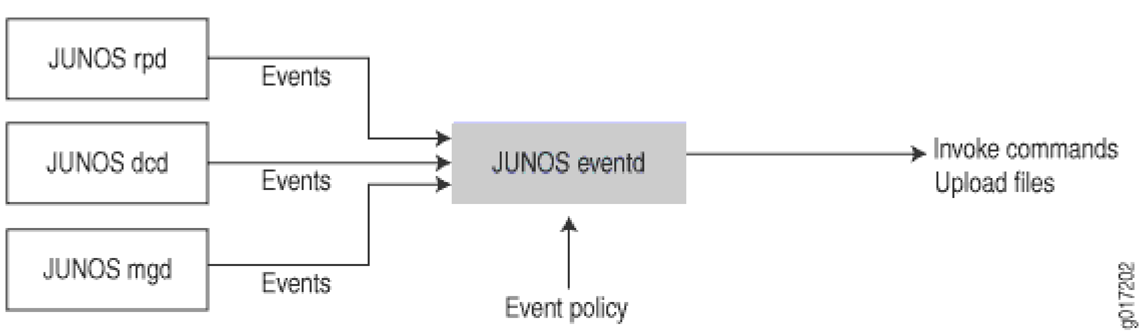
NOTE: In Junos OS Evolved, the order policies appear in may not be the order in which they are executed. Therefore, the behavior is changed so that if one policy has the `ignore` option, none of the policies associated with the event are executed.

NOTE: In Junos OS, eventd throws commit time warning messages if there are duplicate policies. In Junos OS Evolved, eventd will not throw any warning messages. It will accept the policy on a first-come, first-serve basis.

Defining Events

Events can originate as SNMP traps or system log messages. The event process receives event messages from other Junos processes, such as the routing protocol process (*rpd*) and the management process (*mgd*). Figure 8 on page 962 shows how the event process (eventd) interacts with other Junos processes.

Figure 8: Interaction of eventd Process with Other Junos Processes



When you create an event policy, you define one or more events that trigger the policy. There are a number of ways to determine the events to use in a policy. You can explore events by using the following resources:

- System Log Explorer
- `help syslog` operational mode command in the CLI
- context sensitive help when you configure the event policy in the CLI

The System Log Explorer application enables you to search the standard system log messages supported for a given operating system and release.

In the CLI, the `help syslog ?` operational mode command lists the events that you can reference in an event policy.

```
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 CLI User Guide.

You can also list events as you configure the event policy. 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 "Use Nonstandard System Log Messages to Trigger Event Policies" on page 986.

In addition, you can generate internal events at specific times or time intervals and reference those in an event policy. For more information, see "Generate Internal Events to Trigger Event Policies" on page 980.

You can also configure an event policy to trigger for a single event or for two or more correlated events. For information about correlating events, see "Use Correlated Events to Trigger an Event Policy" on page 966.

NOTE: In lab environments, you can the Junos logger utility to simulate one or more events that are difficult to reproduce for a given setup. By simulating the events, you can trigger and test event policies that might be difficult to test otherwise. For more information, see "Junos Logger Utility" on page 987.

Event Policy Actions

You can configure an event policy to execute specific actions in response to events. The event policy can perform one or more of 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 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 configuration mode commands—Execute commands to modify the configuration on receipt of an event. You can configure an event policy to modify the configuration using configuration mode commands and then commit the updated configuration.
- Execute an event script—Execute an event script on receipt of an event. Event scripts are Extensible Stylesheet Language Transformations (*XSLT*), Stylesheet Language Alternative syntaX (SLAX), or Python scripts that you write to perform any function available through Junos XML or Junos XML protocol remote procedure calls (RPCs). For example, a script can run an operational mode command, inspect the command output, and then determine the next appropriate action. This process can be repeated until the source of the problem is determined. The output of the script is stored in a file, which is then uploaded to a specified URL. You can include variables in the arguments to a script that allow data from the triggering event to be incorporated into the script. Additionally, you can define your own set of arguments that is passed to an event script when it’s invoked.
- Raise an SNMP trap.

CHAPTER 35

Event Policy Triggers

IN THIS CHAPTER

Use Correlated Events to Trigger an Event Policy

SUMMARY

Configure an event policy to execute when two or more correlated events occur.

IN THIS SECTION

Understanding Correlated Events

IN THIS SECTION

You can configure an event policy that correlates two or more events. If the 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 correlate events in an event policy, include the following statements at the `[edit event-options]` hierarchy level:

```
[edit event-options]
policy policy-name {
    events [ events ];
    within seconds {
        events [ events ];
        not events [ events ];
        trigger (on | after | until) event-count;
    }
    attributes-match {
        event1.attribute-name equals event2.attribute-name;
        event.attribute-name matches regular-expression;
        event1.attribute-name starts-with event2.attribute-name;
    }
    then {
        ...
    }
}
```

In the `events` statement, you can list multiple trigger events. For information about defining events in event policies, see "Event Policies and Event Notifications Overview" on page 961. To correlate those events with other events, you configure the `within` and/or the `attributes-match` statements.

The `within` statement defines the correlating events that must (or must not) occur within a specified time interval before a trigger event. The `attributes-match` statement correlates an event's attribute with another event's attribute or with a regular expression. The event policy executes the actions configured in the `then` statement only if the specified conditions are met. The following sections discuss how to use the statements.

Correlating Events By Time Interval

You can configure an event policy to execute only if a trigger event occurs within a specified time interval after another event. You do this by configuring the `within` *seconds* `events` statement. The policy is executed if any of the correlating events defined in the `within` *seconds* `events` statement occur within the configured number of seconds before any of the trigger events defined in the first `events` statement. The number of seconds can be from 60 through 604,800. The `not` statement causes the policy to execute only if the correlating events do not occur within the configured time interval before a trigger event.

For example, the device executes the following policy if one of the trigger events, *event3*, *event4*, or *event5*, occurs within 60 seconds after one of the correlating events, *event1* or *event2*, occurs:

```
[edit event-options]
policy 1 {
    events [ event3 event4 event5 ];
    within 60 events [ event1 event2 ];
    then {
        ...
    }
}
```

To configure an event policy to correlate events by time interval:

1. Configure one or more trigger events.

   ```
   [edit event-options policy policy-name]
   user@host# set events [ event3 event4 event 5 ]
   ```

2. Configure the correlating events that must or must not occur before a trigger event and within the specified time interval.

- To specify that a correlating event must occur within the specified time interval before a trigger event, omit the `not` keyword.

```
[edit event-options policy policy-name]
user@host# set within seconds events [ event1 event2 ]
```

- To specify that a correlating event must *not* occur within the specified time interval before a trigger event, include the `not` keyword.

```
[edit event-options policy policy-name]
user@host# set within seconds not events [ event1 event2 ]
```

3. Configure the actions that the event policy executes if the conditions are met.

```
[edit event-options policy policy-name]
user@host# set then ...
```

NOTE: The device executes an event policy if any of the correlating events occur (or do not occur, if you configure the `not` keyword) before any trigger event. To require that multiple correlating events occur (or do not occur) before a trigger event, configure multiple `within` statements. Each statement must specify a different time interval.

The `within` statement also supports executing an event policy when the triggering event occurs a certain number of times within a given time interval. For more information, see "Trigger an Event Policy Based on Event Count" on page 975.

Correlating Events Based on Event Attributes

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 the given regular expression. For more information, see "Use Regular Expressions to Refine the Set of Events That Trigger a Policy" on page 977.
- *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*.

If the `attributes-match` statement includes the `equals` or `starts-with` option, 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, then you must also define one or more `within` statements in the same event policy.

You can use event policy variables within the `attributes-match` statement to differentiate between a triggering event attribute and a correlating event attribute. Trigger events are those that you configure at the `[edit event-options policy` *`policy-name`* `events]` hierarchy level. The double dollar sign (`$$`) notation represents the event that is triggering a policy, and `{$$.`*`attribute-name`*`}` resolves to the value of the attribute of the triggering event. For correlating events, the single dollar sign with the event name (`$`*`event`*) notation represents the most recent event that matches the event name, and `{$`*`event.attribute-name`*`}` resolves to the value of the attribute associated with that event.

For example, the following event policy executes the actions under the `then` statement if four or more commits are performed within a 5-minute period and the username of one or more of the correlating events is the same as the username of the trigger event.

```
policy multiple-commits {
    events ui_commit;
    attributes-match {
        {$$.user-name} equals {$ui_commit.user-name};
    }
    within 300 {
        trigger after 3;
        events ui_commit;
    }
    then ...
}
```

There are many ways to find the attributes that you can reference for a specific event, for example:

- Use System Log Explorer.
- Use the `help syslog` *`event`* operational mode command in the CLI.
- Use context-sensitive help in configuration mode when you configure the attribute.

The System Log Explorer application enables you to search the standard system log messages for a given operating system and release. The message details include the attributes that you can reference for that event.

Alternatively, in the CLI, the `help syslog` *`event`* operational mode command also displays a list of the attributes that you can reference for a given event. The command output shows the event attributes in angle brackets (<>). The following output shows that the `ACCT_ACCOUNTING_SMALL_FILE_SIZE` event has three attributes that you can reference: `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>)
```

NOTE: You can filter the output of a search by using the pipe (|) symbol. For more information about using the pipe symbol, see the CLI User Guide.

You also view event attributes by issuing the `set attributes-match` *`event`*`?` configuration mode 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 (`?`).

How to Represent Triggering and Correlating Events in an Event Policy

In event script arguments and supported event policy statements such as the `execute-commands` statement, you can use event policy variables to differentiate between a *triggering event* and a *correlating event*. Triggering and correlating events are configured in the following statements at the `[edit event-options policy` *`policy-name`*`]` hierarchy level:

- Triggering event—Configured in the `events` statement
- Correlating event—Configured in the `within` *`seconds`* `events` statement

You can use event policy variables of the following forms to represent triggering and correlating events:

- `{$$.`*`attribute-name`*`}`—The double dollar sign (`$$`) notation represents the event that triggers the policy. When combined with an attribute name, the variable resolves to the value of the attribute associated with the triggering event. For example, `{$$.interface-name}` resolves to the interface name associated with the triggering event.

- `{$`*`event.attribute-name`*`}`—The single dollar sign with the event name (`$`*`event`*) notation represents the most recent event that matches *`event`*. When combined with an attribute name, the variable resolves to the value of the attribute associated with that 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 resolves to the interface name associated with the most recent `COSD_CHAS_SCHED_MAP_INVALID` event cached by the event process.
- `{$*.`*`attribute-name`*`}`—The dollar sign with the asterisk (`$*`) notation represents the most recent event that matches any of the correlating events. The variable resolves to the value of the attribute associated with most recent event that matches any of the correlated events specified in the policy configuration.

In event policies, you can reference specific events by using event policy variables. Consider the following 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 `e1`, `e2`, or `e3` occurs within 60 seconds after e4, e5, or e6, 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 e1 occurs within 60 seconds of both e4 and e5, 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: Correlating Events Based on Receipt of Other Events Within a Specified Time Interval

The event policy in this example issues a set of commands and uploads the resulting output file to an archive site. The policy executes if one of the trigger events, *event3*, *event4*, or *event5*, occurs within 60 seconds after one of the correlating events, *event1* or *event2*, occurs. The pseudocode for the policy is as follows:

```
if trigger 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;
}
```

The event policy specifies 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. The event policy configuration is:

```
[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 {
            scp://robot@my.big.com/a/b;
            scp://robot@my.little.com/a/b;
        }
    }
}
```

Example: Correlating Events Based on Event Attributes

In the following event policy, the two events are correlated if their event attribute values match. Matching on the 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 ...
    }
}
```

RELATED DOCUMENTATION

Trigger an Event Policy Based on Event Count | 975

Use Regular Expressions to Refine the Set of Events That Trigger a Policy | 977

Trigger 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 a configuration example, see "Example: Trigger an Event Policy Based on Event Count" on page 975.

Example: Trigger an Event 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 54 on page 976 shows how events add to the count.

Table 54: 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 54 on page 976 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;
        }
    }
}
```

Use 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 that 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 55 on page 978 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 55: 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 Junos OS.

Table 55: Regular Expression Operators for the matches Statement *(Continued)*

Operator	Matches
^ (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.

RELATED DOCUMENTATION

Example: Controlling Event Policy Using a Regular Expression | 979

Example: 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;
        }
    }
}
```

Generate Internal Events to Trigger Event Policies

IN THIS SECTION

Understanding Internal Events

Internal events are events that you create to trigger an event policy. Internal events are not generated by Junos OS processes, and they do not have any associated system log messages. You can configure up to 10 internal events, and you can generate an internal event based on a time interval or the time of day. Event policies can match on internal events in the same way that they match on other events.

NOTE: If you attempt to commit a configuration with more than 10 internal events, Junos OS generates an error, and the commit fails.

To configure an internal event, include the following statements at the `[edit event-options]` hierarchy level:

```
[edit event-options]
generate-event {
    event-name (time-interval seconds <start-time start-time> | time-of-day time) <no-drift>;
}
```

You can configure the `time-interval` option to generate events at a specific frequency, or you can configure the `time-of-day` option to generate events at a specific time of day. Starting in Junos OS Release 14.1, you can configure the `no-drift` option for internal events. When you configure `no-drift`, Junos OS does not propagate the delay caused in triggering an event to the triggering of the next event.

The `start-time` option enables you to specify the start date and time for interval-based events. Thus, the `start-time` option must be configured with the `time-interval` option. If you configure the `start-time` option by itself or with the `time-of-day` option and commit the configuration, the device generates an error, and the commit fails. For example:

```
[edit event-options generate-event event-midnight start-time]
  'start-time "2020-9-1.16:50:00 -0700"'
    'time-interval' is mandatory for 'start-time' configuration
error: commit failed: (statements constraint check failed)
```

How to Generate Events at a Specific Time of Day

To generate an event at a specific time of day, configure the `time-of-day` statement, and specify the time at which the event will occur. A `time-of-day` event is relative to the local device time and is configured using 24-hour time format *hh:mm:ss*. You can optionally include a UTC offset to specify a time relative to UTC (Coordinated Universal Time). The syntax is *hh:mm:ss*(+|-)*hhmm*, where the sign is plus (+) for east of UTC and minus (-) for west of UTC.

NOTE: Coordinated Universal Time is the time on the 0° longitude meridian, also known as the Greenwich meridian. The standard time in most time zones is an integral number of hours offset from UTC.

If you omit the UTC offset, Junos OS automatically generates the offset for the `time-of-day` statement in the configuration based on the local time zone setting for the device. For example, consider a device that uses Eastern Standard Time (UTC−05:00) for its local time. The following command configures an internal event called `event-midnight`, which is generated at 12:00 AM (00:00:00) every night.

```
[edit]
user@host# set event-options generate-event event-midnight time-of-day 00:00:00
```

The resulting configuration automatically includes the device's offset from UTC.

```
user@host# show event-options
generate-event {
    event-midnight time-of-day "00:00:00 -0500";
}
```

If you configure the time relative to UTC, and the offset is different from that of the device, Junos OS automatically converts the time to reflect the device's local time and offset from UTC in the configuration. The following example configures an additional internal event called `event-midnight-cst` that is generated every night at 12:00 AM (00:00:00) with an offset that is six hours behind UTC.

```
[edit]
user@host# set event-options generate-event event-midnight-cst time-of-day 00:00:00-0600
```

The resulting configuration displays the event using the device's local time and UTC offset.

```
user@host# show event-options
generate-event {
    event-midnight time-of-day "00:00:00 -0500";
    event-midnight-cst time-of-day "01:00:00 -0500";
}
```

If the same device is shipped to a different location, and it is configured to use a new time zone, any configured internal events reflect the new local time and time-zone offset. For example:

```
[edit]
user@host# set system time-zone America/Los_Angeles
user@host# commit
```

```
user@host# show event-options
generate-event {
```

```
    event-midnight time-of-day "21:00:00 -0800";
    event-midnight-cst time-of-day "22:00:00 -0800";
}
```

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 the device to generate an internal event 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.

Event policies can match on internal events in the same way that they match on other events. The following example generates an internal event called `it-is-midnight` at 12:00 AM (00:00:00) every night. When the eventd process receives the `it-is-midnight` event, it triggers the `midnight-chores` event policy, which takes certain actions.

```
[edit event-options]
generate-event {
    it-is-midnight time-of-day "00:00:00 -0500";
}
policy midnight-chores {
    events it-is-midnight;
    then {
        ... actions ...
    }
}
```

How to Generate Events at Repeated Time Intervals

You can generate an event at repeated intervals, for example, every hour, by configuring the `time-interval` statement and specifying the interval frequency in seconds. The time interval can range from 60 through 2,592,000 seconds. Junos OS generates the first event starting at approximately one time interval after you commit the configuration.

For example, the following command configures an event called `event-every-hour` that gets generated every hour:

```
[edit]
user@host# set event-options generate-event event-every-hour time-interval 3600
```

The following event policy takes certain actions every 3600 seconds when the `event-every-hour` event is generated:

```
[edit event-options]
generate-event {
    event-every-hour time-interval 3600;
}
policy hourly-checks {
    events event-every-hour;
    then {
        ... actions ...
    }
}
```

On supported devices, you can optionally configure the date and time at which Junos OS starts generating interval-based events. Configuring a start time enables the device to generate events at predictable times, for example, even after a device reboot. To specify a start time for an event, configure the `start-time` option in addition to the `time-interval` option. The start time is relative to the local device time and is configured using 24-hour time format. The syntax is: *yyyy-mm-dd.hh:mm*. For example:

```
[edit event-options generate-event]
user@host# set every-fifteen-minutes time-interval 900 start-time 2020-09-01.16:50
```

If you do not configure the `start-time` option, Junos OS generates the first event at approximately one time interval after you commit the configuration, depending on the time required for the commit operation to complete. If you configure a start time that occurs after the commit time, Junos OS generates the first event at the configured start time, and the next event at one time interval after the start time, and so on. If you configure a start time that is chronologically before the commit time, then Junos OS generates the first event at the next time that is an integral number of time intervals after the start time. Table 56 on page 985 compares the different scenarios for the initial event times based on the start time, if configured, and the time at which the configuration is committed.

Table 56: start-time Commit Examples

`time-interval` (seconds)	`start-time`	Commit time	First event	Second Event
900	–	2020-09-01.17:00:08	2020-09-01.17:15:30	2020-09-01.17:30:30
900	2020-09-01.17:10	2020-09-01.17:00:08	2020-09-01.17:10:00	2020-09-01.17:25:00
900	2020-09-01.16:50	2020-09-01.17:00:08	2020-09-01.17:05:00	2020-09-01.17:20:00
900	2020-09-01.16:50	2020-09-01.17:17:08	2020-09-01.17:20:00	2020-09-01.17:35:00

Similarly, if you configure a start time, Junos OS still generates the next event based on the configured start time even if you reboot the device or restart the event process. Table 57 on page 985 compares the initial event times after the device is rebooted. As shown in the table, when you specify a start time, the device predictably generates the next event as an integral number of time intervals after the start time. When you do not specify a start time, the next event is one time interval after the device comes back online and starts the event process. In this case, it took several minutes for the device to reboot and start the Junos OS processes, so the event occurs well after the time for the next expected event.

Table 57: start-time Reboot Examples

`time-interval` (seconds)	`start-time`	First event	Reboot time	Next Event
900	–	2020-09-01.16:50:00	2020-09-01.16:55:00	2020-09-01.17:13:08
900	2020-09-01.16:50	2020-09-01.16:50:00	2020-09-01.16:55:00	2020-09-01.17:05:00

You can optionally include a UTC offset to specify a start time relative to UTC by appending `(+|-)`*`hhmm`* to the time, where the sign is plus (+) for east of UTC and minus (-) for west of UTC, for example, 2020-09-14.11:00-0800. Even if you specify a UTC offset, the `start-time` configuration always displays the time and UTC offset for the device's local time zone.

The following example configures the start time with a UTC offset:

```
[edit event-options generate-event]
user#host# set every-twenty-minutes time-interval 1200 start-time 2020-09-16.20:30-0200
```

The resulting configuration displays the event time and UTC offset for the device's local time zone.

```
[edit event-options generate-event]
user@host# show
every-twenty-minutes time-interval 1200 start-time "2020-9-16.17:30:00 -0500";
```

Use 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 58 on page 986.

Table 58: Event ID by System Log Message Origin

Event IDs	Origin
SYSTEM	Messages from Junos daemons and utilities
KERNEL	Messages from the kernel
PIC	Messages from Physical Interface Cards (PICs) and Flexible PIC Concentrators (FPCs)
LCC	On a TX Matrix router, messages from a 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 58 on page 986, 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 example, the following event policy references a `KERNEL` system log message. The event policy executes the `raise-trap` action 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;
    }
}
```

Junos Logger Utility

SUMMARY

Use the Junos logger utility to record custom system log messages or generate event notifications for validating event policies in a test environment.

IN THIS SECTION

The Junos software includes a logger utility, which enables you to:

- Record custom messages in the system log file
- Simulate Junos event notifications

Junos Logger Overview

The Junos logger is a shell utility that you can use to log custom messages to the system log file and artificially generate event notifications. The logger enables you to successfully test event policies and event scripts even when the trigger event is difficult to reproduce.

You can configure event policies on Junos devices to execute specific actions in response to an event or series of events. It is important to test an event policy before deploying it in a production environment to ensure that the policy triggers as expected and performs the correct actions. In some cases, it is difficult to simulate the environment that generates the events required to trigger an event policy. In these cases, you can use the Junos logger utility to generate event notifications to trigger the policy.

NOTE: The Junos event logger is an unsupported shell utility that should not be used on devices in a production environment. However, the utility is well suited for use in lab environments where you are developing and verifying event policies and event scripts.

The logger utility also enables you to record a message in the system log without triggering an event notification. Thus, you can log supplementary information about the system in addition to the messages and events that are automatically logged during normal device operations.

When you use the logger utility to generate event notifications or record custom log messages, the device uses a modified version of the base OS's `logger` utility. The modified utility accepts a specialized set of options that are designed for use with the Junos software.

Junos Logger Syntax (Generate an Event)

IN THIS SECTION

Syntax (Junos OS)

```
logger -e EVENT_ID -a attribute=value -d process -l logical-system-name -p priority "message"
```

Syntax (Junos OS Evolved)

```
eventd_logger -e EVENT_ID -a attribute=value -d process -l logical-system-name -p priority
"message"
```

Description

Generate an event notification and also log the event in the system log, provided that the device is configured to log events with the specified priority.

Options

Table 59 on page 990 outlines the required and optional arguments for the Junos event logger utility.

Table 59: Logger Options (Generate an Event)

Option	Description	Example
-a *attribute=value*	(Optional) Attribute associated with the event. The attribute name must be lowercase. To specify multiple attributes, include the -a option for each attribute. For more information about the attributes associated with a particular Junos event, view the event's details in the System Log Explorer tool.	-a username=admin -a command=commit
-d *process*	(Optional) Junos process (daemon) to use as the event's source process. If you do not specify a process, Junos OS uses logger for the process, and Junos OS Evolved uses EVENTD_LOGGER for the process.	-d mgd
-e *EVENT_ID*	Event for which to generate the event notification. The event identifier must be uppercase.	-e UI_COMMIT
-l *logical-system-name*	(Optional) Generate and log an event notification in the log file for the specified logical system.	-l ls-router1
-p *priority*	(Optional) Log the event with the given priority, specified numerically or as a syslog *facility.severity* pair. The default priority is user.notice.	-p external.notice
message	(Optional) Message string to record in the system log. Quotes are not required, but they are recommended for clarity.	"This is a manually generated test event."

Junos Logger Syntax (Log a Message)

IN THIS SECTION

Syntax (Junos OS)

```
logger -46Ais -f file -h host -p priority -t tag "message"
```

Syntax (Junos OS Evolved)

```
eventd_logger -is -f file -p priority -t tag "message"
```

Description

Log entries in the system log, provided that the device is configured to log messages with the specified priority.

NOTE: On devices running Junos OS Evolved, you can also use the Linux logger utility in the system shell to record messages in the system log.

Options

When you use the Junos logger utility to record custom messages in the system log, the device calls the `logger` utility for the underlying base OS. Only a subset of the `logger` options are supported on Junos devices. Table 60 on page 991 outlines the supported options.

Table 60: Logger Options (Log a Message)

Option	Description
`(-4 | -6)`	(Optional) On Junos OS, use IPv4 or IPv6 addresses only.
`-A`	(Optional) On Junos OS, attempt to send the message to all addresses.

Table 60: Logger Options (Log a Message) ***(Continued)***

Option	Description
`-f` *`file`*	(Optional) Copy the contents of the specified file into the system log file.
`-h` *`host`*	(Optional) On Junos OS, send the message to the specified remote host instead of logging it locally.
`-i`	(Optional) Log the process ID of the logger process with each line.
`-p` *`priority`*	(Optional) Generate the event with the given priority, specified numerically or as a syslog *facility.severity* pair. The default priority is `user.notice`.
`-s`	(Optional) Log the message to standard error, in addition to the system log file.
`-t` *`tag`*	(Optional) Log the entry with the specified tag instead of the default tag, which is the current username.
message	(Optional) Message string to record in the system log. Quotes are not required, but they are recommended for clarity.

Use the Junos Logger to Generate Events

The Junos event logger is a shell utility. To use the utility, you must first start a system shell.

To generate an event notification and also log it in the system log file:

1. Start a system shell.

   ```
   user@host> start shell
   ```

   The shell prompt varies depending on the OS and whether the user is the root user. For example:

- In Junos OS:

```
user@host> start shell
%
```

- In Junos OS Evolved:

```
user@host> start shell
[vrf:none] user@host~$
```

2. Generate an event by calling the Junos logger utility and specifying the event ID in uppercase as well as any other optional arguments.

   For example, the following commands simulate a UI_COMMIT event. The event is generated by the `mgd` process and has event priority `external.notice`.

   - In Junos OS:

   ```
   % logger -e UI_COMMIT -d mgd -p external.notice -a username=admin -a command=commit "This is a test event."
   ```

   - In Junos OS Evolved:

   ```
   [vrf:none] user@host~$ eventd_logger -e UI_COMMIT -d mgd -p external.notice -a username=admin -a command=commit "This is a test event."
   ```

The event notification is sent to the `eventd` process. The device also records the event information and specified message in the system log file, provided you have configured the device to log messages of the given facility and severity level.

```
user@host> show log messages | match UI_COMMIT
Nov 4 16:26:41 host mgd: UI_COMMIT: This is a test event.
```

You can use the Junos event logger utility to generate one or more event notifications to trigger and test an event policy when it is difficult to reproduce the events in a given lab environment. In the previous example, the device will execute any policies that trigger on the UI_COMMIT event.

Use the Junos Logger to Log Messages

The Junos logger is a shell utility. To use the utility, you must first start a system shell.

To log information in the system log file:

1. Start a system shell.

```
user@host> start shell
```

The shell prompt varies depending on the OS and whether the user is the root user. For example:

- In Junos OS:

```
user@host> start shell
%
```

- In Junos OS Evolved:

```
user@host> start shell
[vrf:none] user@host~$
```

2. Log a message to the system log by calling the Junos logger utility and specifying the information to log as well as any optional arguments.

For example, the following commands log the given message string with the `CUSTOM_LOG` tag and a priority of `external.notice`.

- In Junos OS:

```
% logger -p external.notice -t CUSTOM_LOG "The server is up."
```

- In Junos OS Evolved:

```
[vrf:none] user@host~$ eventd_logger -p external.notice -t CUSTOM_LOG "The server is up."
```

In this example, the message string is recorded in the system log with the CUSTOM_LOG tag, provided you have configured the device to log messages of the given facility and severity level.

```
user@host> show log messages | match CUSTOM_LOG
Nov 16 14:38:31  host CUSTOM_LOG[21905]: The server is up.
```

CHAPTER 36

Event Policy Actions

IN THIS CHAPTER

Configure an Event Policy to Execute Operational Mode Commands

Operational mode commands perform an operation or provide diagnostic output on a device running Junos OS. They enable you to view statistics and information about a device's current operating status. They also enable you to take corrective actions, such as restarting software processes, taking a Physical Interface Card (*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 CLI Explorer.

You can configure an event policy that executes operational mode commands and uploads the output of those commands to a specified location for analysis by including 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 eventd process executes the operational mode commands configured for the commands statement. 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 triggers the policy. When combined with an attribute name, the variable resolves to the value of the attribute associated with the triggering event. For example, {$$.interface-name} resolves to the interface name associated with the triggering event.

- `{$event.attribute-name}`—The single dollar sign with the event name (`$event`) notation represents the most recent event that matches *event*. When combined with an attribute name, the variable resolves to the value of the attribute associated with that 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 resolves to the interface name associated with the most recent `COSD_CHAS_SCHED_MAP_INVALID` event cached by the event process.

- `{$*.attribute-name}`—The dollar sign with the asterisk (`$*`) notation represents the most recent event that matches any of the correlating events. The variable resolves to the value of the attribute associated with most recent event that matches any of the correlated events specified in the policy configuration.

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` command:

```
user@host> help syslog event
```

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
...
```

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 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
  ...
```

When the eventd process executes the commands, it uploads the file with the command output to the location specified in the `destination` statement. In the `destination` statement, include a destination name that is configured at the `[edit event-options destinations]` hierarchy level. For more information, see "Example: Define Destinations for File Archiving by Event Policies" on page 1062.

In the `output-filename` statement, define a descriptive string that will be included in the filename. Each uploaded file also includes the hostname and timestamp in the filename to ensure that the each filename is unique. If a policy is triggered multiple times in a 1-second period, an index number is appended to the filename to ensure that the filenames are still unique. The index number range is 001 through 999.

Starting in Junos OS Release 14.1R3, the naming convention and format for the output file generated by eventd are changed. When eventd executes the commands defined at the `[edit event-options policy *policy-name* then execute-commands commands]` hierarchy level and generates a file containing the command output, the name of the file depends on the version of Junos OS running on the device. Prior to Junos OS Release 14.1R3, the filename has the following naming convention:

```
hostname_output-filename_YYYYMMDD_HHMMSS_index-number
```

Starting in Junos OS Release 14.1R3, the filename places the ***output-filename*** string after the timestamp.

```
hostname_YYYYMMDD_HHMMSS_output-filename_index-number
```

For example, on a device named r1 running Junos OS Release 14.1R3 or a later release, if you configure the `output-filename` statement as **ifl-events**, and this event policy is triggered three times in 1 second, the files are named:

- **r1_20060623_132333_ifl-events**
- **r1_20060623_132333_ifl-events_001**
- **r1_20060623_132333_ifl-events_002**

By default, the command output format is Junos Extensible Markup Language (XML). Configure the `output-format text` statement to format the command output as *ASCII* text.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
14.1R3	Starting in Junos OS Release 14.1R3, the naming convention and format for the output file generated by eventd are changed.

RELATED DOCUMENTATION

Example: Correlating Events Based on Receipt of Other Events Within a Specified Time Interval | 973

Configure an Event Policy to Change the Configuration

An event policy performs actions in response to specific events. You can configure custom event policies in the Junos OS configuration that listen for a specific event or correlated events and then execute an action, which might include creating a log file, invoking Junos OS commands, or executing an event script. At times, it might be necessary to modify the configuration in response to a particular event. For example, upon receipt of an SNMP_TRAP_LINK_DOWN or SNMP_TRAP_LINK_UP event for a given interface, the event policy action might modify the configuration of a static route to adjust its metric or modify its next hop.

Event policies can modify the configuration by invoking an event script that changes and commits the configuration or by using the `change-configuration` statement to execute configuration mode commands that change the configuration. Prior to Junos OS Release 12.1, an event policy invoked an event script to execute configuration changes. Starting in Junos OS Release 12.1, in addition to invoking an event script, you can configure an event policy with the `change-configuration` action, which uses Junos OS configuration mode commands to modify the configuration.

> **NOTE**: Do not use the `change-configuration` statement to modify the configuration on dual Routing Engine devices that have nonstop active routing (NSR) enabled, because both Routing Engines might attempt to acquire a lock on the configuration database, which can cause the commit to fail. To modify the configuration through an event policy when NSR is enabled, invoke an event script that executes the commit on only the primary Routing Engine.

To modify the configuration through an event policy using configuration mode commands, configure the `change-configuration` statement at the `[edit event-options policy policy-name then]` hierarchy level. For example:

```
[edit event-options policy policy-name then]
change-configuration {
    commands {
        "set routing-options static route 198.51.100.0/24 next-hop 10.1.3.1";
    }
}
```

The `commands` statement specifies the configuration mode commands that are executed upon receipt of the configured event or events. Enclose each command in quotation marks (`" "`), and specify the complete statement path to the element, identifier, or value as you do in configuration mode when issuing commands at the `[edit]` hierarchy level. The `commands` statement accepts the following configuration mode commands:

- `activate`
- `deactivate`
- `delete`
- `set`

The event process (eventd) executes the configuration commands in the order in which they appear in the event policy configuration. The commands update the candidate configuration, which is then committed, provided that no commit errors occur.

You can configure the `commit-options` child statement to customize the event policy commit operation. You can commit the changes on a single Routing Engine or configure the `synchronize` option to synchronize the commit on both Routing Engines. When you configure the `synchronize` option, the Routing Engine on which you execute this command copies and loads its candidate configuration to the other Routing

Engine. Both Routing Engines perform a syntax check on the candidate configuration file. If no errors are found, the configuration is activated and becomes the current operational configuration on both Routing Engines. By default, the `synchronize` option does not work if the responding Routing Engine has uncommitted configuration changes. However, you can enforce commit synchronization on the Routing Engines and ignore any warnings by configuring the `force` option.

Additionally, if you are testing or troubleshooting an event policy, you can configure the `check` commit option to verify the candidate configuration syntax without committing the changes. On dual control plane systems, when the `check synchronize` statement is configured, the candidate configuration on one control plane is copied to the other control plane, and the system verifies that both candidate configurations are syntactically correct. The `check` statement and the other `commit-options` statements are mutually exclusive.

The change configuration action might fail while acquiring a lock on the configuration. Configure the `retry` statement to have the system attempt the change configuration event policy action a specified number of times if the first attempt fails. Configure the `user-name` statement to execute the configuration changes and commit under the privileges of a specific user. If you do not specify a username, the action is executed as user `root`.

RELATED DOCUMENTATION

Example: Changing the Configuration Using an Event Policy | 1001
Example: Changing the Interface Configuration in Response to an Event | 1011
Change the Configuration Using an Event Script | 1025
change-configuration (Event Policy)

Example: Changing the Configuration Using an Event Policy

IN THIS SECTION

It might be necessary to modify the configuration in response to a particular event. Starting in Junos OS Release 12.1, you can configure an event policy to make and commit configuration changes when the event policy is triggered by one or more specific events.

This example simulates an SNMP_TRAP_LINK_DOWN event for a specific interface. Upon receipt of the event, the event policy uses the `change-configuration` action to modify the configuration of a static route to use a new next-hop IP address through a different exit interface.

Requirements

- Routing, switching, or security device running Junos OS Release 12.1 or later.

Overview

You can configure an event policy action to modify the configuration when the policy is triggered by a single event or correlated events. Suppose you have a static route to the 10.1.10.0/24 network with a next-hop IP address of 10.1.2.1 through the exit interface ge-0/3/1. At some point, this interface goes down, triggering an SNMP_TRAP_LINK_DOWN event.

This example creates an event policy named update-on-snmp-trap-link-down. The event policy is configured so that the eventd process listens for an SNMP_TRAP_LINK_DOWN event associated with the interface ge-0/3/1.0. If the interface goes down, the event policy executes a change configuration action. The event policy configuration commands remove the static route through the ge-0/3/1 exit interface and create a new static route to the same target network with a next-hop IP address of 10.1.3.1 through the exit interface ge-0/2/1. The commands are executed in the order in which they appear in the event policy.

The event policy change configuration commit operation is executed under the username bsmith with a commit comment specifying that the change was made through the associated event policy. The retry count is set to 5 and the retry interval is set to 4 seconds. If the initial attempt to issue the configuration change fails, the system attempts the configuration change 5 additional times and waits 4 seconds between each attempt.

Although not presented here, you might have a second, similar event policy that executes a change configuration action to update the static route when the interface comes back up. In that case the policy would trigger on the SNMP_TRAP_LINK_UP event for the same interface.

Configuration

IN THIS SECTION

CLI Quick Configuration

To quickly configure this example, copy the following commands, paste them in a text file, remove any line breaks, change any details necessary to match your network configuration, and then copy and paste the commands into the CLI at the `[edit]` hierarchy level:

```
set event-options policy update-on-snmp-trap-link-down events snmp_trap_link_down
set event-options policy update-on-snmp-trap-link-down attributes-match
snmp_trap_link_down.interface-name matches ge-0/3/1.0
set event-options policy update-on-snmp-trap-link-down then change-configuration retry count 5
set event-options policy update-on-snmp-trap-link-down then change-configuration retry interval 4
set event-options policy update-on-snmp-trap-link-down then change-configuration commands
"delete routing-options static route 10.1.10.0/24 next-hop"
set event-options policy update-on-snmp-trap-link-down then change-configuration commands "set
routing-options static route 10.1.10.0/24 next-hop 10.1.3.1"
set event-options policy update-on-snmp-trap-link-down then change-configuration user-name bsmith
set event-options policy update-on-snmp-trap-link-down then change-configuration commit-options
log "updating configuration from event policy update-on-snmp-trap-link-down"
set routing-options static route 10.1.10.0/24 next-hop 10.1.2.1
set system syslog file syslog-event-daemon-warning daemon warning
```

Configuring the Event Policy

Step-by-Step Procedure

1. Create and name the event policy.

```
[edit]
bsmith@R1# edit event-options policy update-on-snmp-trap-link-down
```

2. Configure the `events` statement so that the event policy triggers on the SNMP_TRAP_LINK_DOWN event.

Set the `attributes-match` statement so that the policy triggers only if the SNMP_TRAP_LINK_DOWN event occurs for the ge-0/3/1.0 interface.

```
[edit event-options policy update-on-snmp-trap-link-down]
bsmith@R1# set events snmp_trap_link_down
bsmith@R1# set attributes-match snmp_trap_link_down.interface-name matches ge-0/3/1.0
```

3. Specify the configuration mode commands that are executed if the ge-0/3/1 interface goes down.

   Configure each command on a single line, enclose the command string in quotes, and specify the complete statement path.

```
[edit event-options policy update-on-snmp-trap-link-down then change-configuration]
bsmith@R1# set commands "delete routing-options static route 10.1.10.0/24 next-hop"
bsmith@R1# set commands "set routing-options static route 10.1.10.0/24 next-hop 10.1.3.1"
```

4. Configure the commit options.

   Configure the `log` option with a comment describing the configuration changes. The comment is added to the commit logs after a successful commit operation is made through the associated event policy.

```
[edit event-options policy update-on-snmp-trap-link-down then change-configuration]
bsmith@R1# set commit-options log "updating configuration from event policy update-on-snmp-
trap-link-down"
```

   If you have dual Routing Engines, configure the `synchronize` option to commit the configuration on both Routing Engines. Include the `force` option to force the commit on the other Routing Engine, ignoring any warnings. This example does not configure the `synchronize` and `force` options.

5. (Optional) Configure the retry count and retry interval.

   In this example, `count` is set to 5 and the `interval` is 4 seconds.

```
[edit event-options policy update-on-snmp-trap-link-down then change-configuration]
bsmith@R1# set retry count 5 interval 4
```

6. (Optional) Configure the username under whose privileges the configuration changes and commit are made.

If you do not specify a username, the action is executed as user root.

```
[edit event-options policy update-on-snmp-trap-link-down then change-configuration]
bsmith@R1# set user-name bsmith
```

7. Configure a new log file at the `[edit system syslog]` hierarchy level to record syslog events of facility `daemon` and severity `warning`.

   This captures the SNMP_TRAP_LINK_DOWN events.

```
[edit system syslog]
bsmith@R1# set file syslog-event-daemon-warning daemon warning
```

8. To test this example, configure a static route to the 10.1.10.0/24 network with a next hop IP address of 10.1.2.1.

```
[edit]
bsmith@R1# set routing-options static route 10.1.10.0/24 next-hop 10.1.2.1
```

9. Commit the configuration.

```
bsmith@R1# commit
```

10. Review the `[edit routing-options static]` hierarchy level of the configuration before disabling the ge-0/3/1 interface, and note the next hop IP address.

```
bsmith@R1> show configuration routing-options static
...
route 10.1.10.0/24 next-hop 10.1.2.1;
...
```

11. To manually test the event policy, take the ge-0/3/1 interface temporarily offline to generate the SNMP_TRAP_LINK_DOWN event.

```
[edit]
bsmith@R1# set interfaces ge-0/3/1 disable
bsmith@R1# commit
```

Results

```
[edit]
event-options {
    policy update-on-snmp-trap-link-down {
        events snmp_trap_link_down;
        attributes-match {
            snmp_trap_link_down.interface-name matches ge-0/3/1.0;
        }
        then {
            change-configuration {
                retry count 5 interval 4;
                commands {
                    "delete routing-options static route 10.1.10.0/24 next-hop";
                    "set routing-options static route 10.1.10.0/24 next-hop 10.1.3.1";
                }
                user-name bsmith;
                commit-options {
                    log "updating configuration from event policy update-on-snmp-trap-link-down";
                }
            }
        }
    }
}
routing-options {
    static {
        route 10.1.10.0/24 next-hop 10.1.2.1;
    }
}
system {
    syslog {
        file syslog-event-daemon-warning {
            daemon warning;
        }
    }
}
```

Verification

IN THIS SECTION

Confirm that the configuration is working properly.

Verifying the Status of the Interface

Purpose

Verify that the ge-0/3/1 interface is down and that it triggered the SNMP_TRAP_LINK_DOWN event.

Action

Issue the `show interfaces ge-0/3/1` operational mode command. The command output shows that the interface is administratively offline.

```
bsmith@R1> show interfaces ge-0/3/1
Physical interface: ge-0/3/1, Administratively down, Physical link is Down
<output omitted>
```

Review the contents of the system log file configured in Step "7" on page 1005. The output shows that the ge-0/3/1.0 interface went down and generated an SNMP_TRAP_LINK_DOWN event.

```
bsmith@R1> show log syslog-event-daemon-warning
Oct 10 18:00:57  R1 mib2d[1371]: SNMP_TRAP_LINK_DOWN: ifIndex 531, ifAdminStatus down(2),
ifOperStatus down(2), ifName ge-0/3/1.0
```

Verifying the Commit

Purpose

Verify that the event policy commit operation was successful by reviewing the commit log and the messages log file.

Action

Issue the `show system commit` operational mode command to view the commit log. In this example, the log confirms that the configuration was committed through the event policy under the privileges of user bsmith at the given date and time.

```
bsmith@R1> show system commit
0   2011-10-10 18:01:03 PDT by bsmith via junoscript
    updating configuration from event policy update-on-snmp-trap-link-down
1   2011-09-02 14:16:44 PDT by admin via netconf
2   2011-07-08 14:33:46 PDT by root via other
```

Review the **messages** log file. Upon receipt of the SNMP_TRAP_LINK_DOWN event, Junos OS executed the configured event policy action to modify and commit the configuration. The commit operation occurred under the privileges of user bsmith.

```
bsmith@R1> show log messages | last 20
...
Oct 10 18:00:57  R1 mib2d[1371]: SNMP_TRAP_LINK_DOWN: ifIndex 531, ifAdminStatus down(2),
ifOperStatus down(2), ifName ge-0/3/1.0
Oct 10 18:00:59  R1 file[17575]: UI_COMMIT: User 'bsmith' requested 'commit' operation (comment:
updating configuration from event policy update-on-snmp-trap-link-down)
Oct 10 18:01:03  R1 eventd: EVENTD_CONFIG_CHANGE_SUCCESS: Configuration change successful: while
executing policy update-on-snmp-trap-link-down with user bsmith privileges
```

NOTE: If you configure a different log file, review the file specific to your configuration.

Meaning

The output from the `show system commit` operational mode command and the **messages** log file verify that the commit operation, which was made through the event policy under the privileges of the user bsmith, was successful. The `show system commit` output and **messages** log file reference the commit comment specified in the `log` statement at the `[edit event-options policy update-on-snmp-trap-link-down then change-configuration commit-options]` hierarchy level.

Verifying the Configuration Changes

Purpose

Verify the configuration changes by reviewing the `[edit routing-options static]` hierarchy level of the configuration after disabling the ge-0/3/1 interface.

Action

Issue the following operational mode command:

```
bsmith@R1> show configuration routing-options static
...
route 10.1.10.0/24 next-hop 10.1.3.1;
...
```

Meaning

The configured next hop has been modified by the event policy to the new IP address 10.1.3.1, which has its route through the exit interface ge-0/2/1.

Troubleshooting

IN THIS SECTION

Troubleshooting Commit Errors

Problem

The triggered event policy does not make the specified configuration changes, and the logs verify that the commit was unsuccessful.

```
bsmith@R1> show log messages | last 20
...
Oct 10 17:48:59  R1 mib2d[1371]: SNMP_TRAP_LINK_DOWN: ifIndex 531, ifAdminStatus down(2),
ifOperStatus down(2), ifName ge-0/3/1.0
Oct 10 17:49:01  R1 file[17142]: UI_LOAD_EVENT: User 'bsmith' is performing a 'rollback'
Oct 10 17:49:01  R1 eventd: EVENTD_CONFIG_CHANGE_FAILED: Configuration change failed: rpc to
management daemon failed while executing policy update-on-snmp-trap-link-down with user bsmith
privileges
```

A failed commit might occur if the configuration is locked or if the configuration mode commands have the incorrect syntax or order.

Solution

Check the configuration mode commands at the `[edit event-options policy update-on-snmp-trap-link-down then change-configuration commands]` hierarchy level, and verify that the syntax and the order of execution are correct.

Additionally, increase the retry count and interval options so that if the configuration is locked, the event policy attempts the configuration changes a specified number of times after the first failed instance.

RELATED DOCUMENTATION

Configure an Event Policy to Change the Configuration | 999

Example: Changing the Interface Configuration in Response to an Event | 1011

change-configuration (Event Policy)

commands (Event Policy Change Configuration)

commit-options

retry (Event Policy)

Example: Changing the Interface Configuration in Response to an Event

IN THIS SECTION

It might be necessary to modify the configuration in response to a particular event. Starting in Junos OS Release 12.1, you can configure an event policy to make and commit configuration changes when the event policy is triggered by one or more specific events.

This example uses a real-time performance monitoring (RPM) probe to generate PING_TEST_FAILED events for a given interface. Upon receipt of the first instance of three PING_TEST_FAILED events within a 60-second period from the configured RPM probe, the event policy executes a change configuration event policy action that modifies the configuration to administratively disable the specified interface. This type of action might be necessary if you have an unstable, flapping interface that is consistently affecting network performance.

Requirements

- Routing, switching, or security device running Junos OS Release 12.1 or later.

Overview

This example creates an event policy named disable-interface-on-ping-failure. The event policy is configured so that the eventd process listens for PING_TEST_FAILED events generated by a specific RPM probe and associated with the ge-0/3/1 interface. If three PING_TEST_FAILED events occur for the given interface within a 60-second interval, the event policy executes a change configuration action. The event policy configuration commands administratively disable the interface.

To test the event policy, the example configures an RPM probe that pings the IP address associated with the ge-0/3/1 interface. In this example, the ge-0/3/1.0 interface is configured with the IPv4 address 10.1.4.1/26. By default, one probe is sent per test, and the example uses a 5-second pause between tests. After three successive probe losses, the RPM probe generates a PING_TEST_FAILED event. Because multiple RPM tests could be running simultaneously, the event policy matches the `owner-name` and `test-name` attributes of the received PING_TEST_FAILED events to the RPM probe owner name and test name. When the RPM probe generates three PING_TEST_FAILED events in a 60-second interval, it triggers the event policy, which disables the interface.

This event policy also demonstrates how to restrict the execution of the same configuration change multiple times because of occurrences of the same event or correlated events. In this example, the `within 60 trigger on 3` statement specifies that the configuration change is only triggered on the third occurrence of a PING_TEST_FAILED event within a 60-second interval. The `trigger until 4` statement specifies that subsequent occurrences of the PING_TEST_FAILED event should not cause the event policy to re-trigger.

NOTE: If you only configure the `trigger on 3` condition, the commit operation might go into a loop. The combination of `trigger on 3` and `trigger until 4` prevents the event policy from repeatedly making the same configuration change.

Configuration

IN THIS SECTION

Configuring the RPM Probe

CLI Quick Configuration

To quickly configure this section of the example, copy the following commands, paste them into a text file, remove any line breaks, change any details necessary to match your network configuration, copy and paste the commands into the CLI at the `[edit]` hierarchy level, and then enter `commit` from configuration mode.

```
set services rpm probe icmp-ping-probe test ping-probe-test probe-type icmp-ping
set services rpm probe icmp-ping-probe test ping-probe-test target address 10.1.4.1
set services rpm probe icmp-ping-probe test ping-probe-test test-interval 5
set services rpm probe icmp-ping-probe test ping-probe-test thresholds successive-loss 3
set system syslog file syslog-event-daemon-info daemon info
```

Step-by-Step Procedure

To configure the RPM probe, which creates the PING_TEST_FAILED events for this example:

1. Create an RPM probe named ping-probe-test with owner icmp-ping-probe.

```
[edit services rpm]
bsmith@R1# set probe icmp-ping-probe test ping-probe-test
```

2. Configure the RPM probe to send ICMP echo requests.

```
[edit services rpm probe icmp-ping-probe test ping-probe-test]
bsmith@R1# set probe-type icmp-ping
```

   Configure the RPM probe to send the ICMP echo requests to the ge-0/3/1 interface at IP address 10.1.4.1.

```
[edit services rpm probe icmp-ping-probe test ping-probe-test]
bsmith@R1# set target address 10.1.4.1
```

3. Configure a 5-second pause between test windows.

```
[edit services rpm probe icmp-ping-probe test ping-probe-test]
bsmith@R1# set test-interval 5
```

4. Configure the RPM probe threshold so that the PING_TEST_FAILED event is triggered after three successive probe losses.

```
[edit services rpm probe icmp-ping-probe test ping-probe-test]
bsmith@R1# set thresholds successive-loss 3
```

5. Configure a new log file at the `[edit system syslog]` hierarchy level to record syslog events of facility `daemon` and severity `info`.

This captures the events sent during the probe tests.

```
[edit system syslog]
bsmith@R1# set file syslog-event-daemon-info daemon info
```

6. Commit the configuration.

```
bsmith@R1# commit
```

Results

From configuration mode, confirm your configuration by entering the `show services` and `show system syslog` commands. If the output does not display the intended configuration, repeat the instructions in this example to correct the configuration.

```
[edit]
services {
    rpm {
        probe icmp-ping-probe {
            test ping-probe-test {
                probe-type icmp-ping;
                target address 10.1.4.1;
                test-interval 5;
                thresholds {
                    successive-loss 3;
                }
            }
        }
    }
}
system {
    syslog {
        file syslog-event-daemon-info {
            daemon info;
        }
    }
}
```

Configuring the Event Policy

CLI Quick Configuration

To quickly configure this section of the example, copy the following commands, paste them into a text file, remove any line breaks, change any details necessary to match your network configuration, copy and paste the commands into the CLI at the `[edit]` hierarchy level, and then enter `commit` from configuration mode.

```
set event-options policy disable-on-ping-failure events ping_test_failed
set event-options policy disable-on-ping-failure within 60 trigger on
set event-options policy disable-on-ping-failure within 60 trigger 3
set event-options policy disable-on-ping-failure within 65 trigger until
set event-options policy disable-on-ping-failure within 65 trigger 4
set event-options policy disable-on-ping-failure attributes-match ping_test_failed.test-owner
matches icmp-ping-probe
set event-options policy disable-on-ping-failure attributes-match ping_test_failed.test-name
matches ping-probe-test
set event-options policy disable-on-ping-failure then change-configuration commands "set
interfaces ge-0/3/1 disable"
set event-options policy disable-on-ping-failure then change-configuration user-name bsmith
set event-options policy disable-on-ping-failure then change-configuration commit-options log
"updating configuration from event policy disable-on-ping-failure"
```

Step-by-Step Procedure

1. Create and name the event-policy.

```
[edit]
bsmith@R1# edit event-options policy disable-interface-on-ping-failure
```

2. Configure the event policy to match on the PING_TEST_FAILED event.

```
[edit event-options policy disable-interface-on-ping-failure]
bsmith@R1# set events ping_test_failed
```

3. Configure the policy to trigger when three PING_TEST_FAILED events occur within 60 seconds.

```
[edit event-options policy disable-interface-on-ping-failure]
bsmith@R1# set within 60 trigger on 3
```

4. Configure the `within 65 trigger until 4` statement to prevent the policy from re-triggering if more than three PING_TEST_FAILED events occur.

```
[edit event-options policy disable-interface-on-ping-failure]
bsmith@R1# set within 65 trigger until 4
```

5. Configure the `attributes-match` statement so that the event policy is only triggered by the PING_TEST_FAILED events generated by the associated RPM probe.

```
[edit event-options policy disable-interface-on-ping-failure]
bsmith@R1# set attributes-match ping_test_failed.test-owner matches icmp-ping-probe
bsmith@R1# set attributes-match ping_test_failed.test-name matches ping-probe-test
```

6. Specify the configuration mode commands that are executed if the event policy is triggered.

   Configure each command on a single line, enclose the command string in quotes, and specify the complete statement path.

```
[edit event-options policy disable-interface-on-ping-failure then change-configuration]
bsmith@R1# set commands "set interfaces ge-0/3/1 disable"
```

7. Configure the `log` option with a comment describing the configuration changes.

   The comment is added to the commit logs after a successful commit operation is made through the associated event policy.

```
[edit event-options policy disable-interface-on-ping-failure then change-configuration]
bsmith@R1# set commit-options log "updating configuration from event policy disable-
interface-on-ping-failure"
```

8. (Optional) If you have dual Routing Engines, configure the `synchronize` option to commit the configuration on both Routing Engines. Include the `force` option to force the commit on the other

Routing Engine, ignoring any warnings. This example does not configure the synchronize and force options.

9. (Optional) Configure the username under whose privileges the configuration changes and commit are made.

   If you do not specify a username, the action is executed as user root.

```
[edit event-options policy disable-interface-on-ping-failure then change-configuration]
bsmith@R1# set user-name bsmith
```

10. Review the output of the show interfaces ge-0/3/1 operational mode command before the configuration change takes place.

   NOTE: The interface should be enabled.

```
bsmith@R1> show interfaces ge-0/3/1
Physical interface: ge-0/3/1, Enabled, Physical link is Up
  Interface index: 142, SNMP ifIndex: 531
...
```

11. Commit the configuration.

```
bsmith@R1# commit
```

Results

From configuration mode, confirm your configuration by entering the show event-options command. If the output does not display the intended configuration, repeat the instructions in this example to correct the configuration.

```
[edit event-options]
policy disable-interface-on-ping-failure {
    events ping_test_failed;
    within 60 {
        trigger on 3;
    }
    within 65 {
```

```
        trigger until 4;
    }
    attributes-match {
        ping_test_failed.test-owner matches icmp-ping-probe;
        ping_test_failed.test-name matches ping-probe-test;
    }
    then {
        change-configuration {
            commands {
                "set interfaces ge-0/3/1 disable";
            }
            user-name bsmith;
            commit-options {
                log "updating configuration from event policy disable-interface-on-ping-failure";
            }
        }
    }
}
```

Verification

IN THIS SECTION

Confirm that the configuration is working properly.

Verifying the Events

Purpose

To manually test the event policy, take the ge-0/3/1 interface offline until three PING_TEST_FAILED events are generated.

Action

Review the configured syslog file. Verify that the RPM probe generates a PING_TEST_FAILED event after successive lost probes.

```
bsmith@R1> show log syslog-event-daemon-info
Oct  7 15:48:54  R1 rmopd[1345]: PING_TEST_COMPLETED: pingCtlOwnerIndex = icmp-ping-probe,
pingCtlTestName = ping-probe-test
Oct  7 15:49:54  R1 rmopd[1345]: PING_TEST_COMPLETED: pingCtlOwnerIndex = icmp-ping-probe,
pingCtlTestName = ping-probe-test
...
Oct  7 15:52:54  R1 rmopd[1345]: RMOPD_ICMP_SENDMSG_FAILURE: sendmsg(ICMP): No route to host
Oct  7 15:52:54  R1 rmopd[1345]: PING_PROBE_FAILED: pingCtlOwnerIndex = icmp-ping-probe,
pingCtlTestName = ping-probe-test
Oct  7 15:52:54  R1 rmopd[1345]: PING_TEST_FAILED: pingCtlOwnerIndex = icmp-ping-probe,
pingCtlTestName = ping-probe-test
Oct  7 15:52:57  R1 rmopd[1345]: PING_TEST_FAILED: pingCtlOwnerIndex = icmp-ping-probe,
pingCtlTestName = ping-probe-test
Oct  7 15:53:00  R1 rmopd[1345]: PING_TEST_FAILED: pingCtlOwnerIndex = icmp-ping-probe,
pingCtlTestName = ping-probe-test
```

Verifying the Commit

Purpose

Verify that the event policy commit operation was successful by reviewing the commit log and the messages log file.

Action

Issue the `show system commit` operational mode command to view the commit log.

```
bsmith@R1> show system commit
0   2011-10-07 15:53:00 PDT by bsmith via junoscript
    updating configuration from event policy disable-interface-on-ping-failure
1   2011-09-02 14:16:44 PDT by admin via netconf
2   2011-07-08 14:33:46 PDT by root via other
```

Review the messages log file.

```
bsmith@R1> show log messages | last 20
Oct  7 15:52:54  R1 rmopd[1345]: RMOPD_ICMP_SENDMSG_FAILURE: sendmsg(ICMP): No route to host
Oct  7 15:53:00  R1 file[9972]: UI_COMMIT: User 'bsmith' requested 'commit' operation (comment:
updating configuration from event policy disable-interface-on-ping-failure)
Oct  7 15:53:02  R1 eventd: EVENTD_CONFIG_CHANGE_SUCCESS: Configuration change successful: while
executing policy disable-interface-on-ping-failure with user bsmith privileges
```

Meaning

The output from the `show system commit` operational mode command and the **messages** log file verify that Junos OS executed the configured event policy action to modify and commit the configuration. The commit operation, which was made through the event policy under the privileges of the user bsmith at the given date and time, was successful. The `show system commit` output and **messages** log file reference the commit comment specified in the `log` statement at the `[edit event-options policy disable-interface-on-ping-failure then change-configuration commit-options]` hierarchy level.

Verifying the Configuration Changes

Purpose

Verify the configuration changes by reviewing the `[edit interfaces ge-0/3/1]` hierarchy level of the configuration.

Action

```
bsmith@R1> show configuration interfaces ge-0/3/1
disable;
unit 0 {
    family inet {
        address 10.1.4.1/26;
    }
}
```

Meaning

The ge-0/3/1 configuration hierarchy was modified through the event policy to add the `disable` statement.

Verifying the Status of the Interface

Purpose

Review the output of the `show interfaces ge-0/3/1` operational mode command after the configuration change takes place.

Action

Issue the `show interfaces ge-0/3/1` operational mode command. After the event policy configuration change action disables the interface, the status changes from "Enabled" to "Administratively down".

```
bsmith@R1> show interfaces ge-0/3/1
Physical interface: ge-0/3/1, Administratively down, Physical link is Down
  Interface index: 142, SNMP ifIndex: 531
```

RELATED DOCUMENTATION

Configure an Event Policy to Change the Configuration | 999

Example: Changing the Configuration Using an Event Policy | 1001

change-configuration (Event Policy)

commands (Event Policy Change Configuration)

commit-options

retry (Event Policy)

Execute Event Scripts in an Event Policy

Event scripts are Extensible Stylesheet Language Transformations (*XSLT*) scripts, Stylesheet Language Alternative syntaX (SLAX) scripts, or Python scripts that an event policy can execute when it is triggered. 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.

Event scripts 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 script can then report the source of the problem to you on the CLI or automatically change the device configuration.

You can configure an event policy that executes event scripts and uploads the output of those scripts 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;
            }
            destination destination-name;
            output-filename filename;
            output-format (text | xml);
        }
    }
}
```

In the `events` statement, you can list multiple events. If one or more of the listed events occurs, the eventd process executes the actions configured under the `then` hierarchy. The event policy actions can include executing one or more event scripts, which are configured by including the `event-script` *`filename`* statement. 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 disk or the **/config/scripts/event/** directory in flash memory Furthermore, the event scripts must be enabled at the `[edit event-options event-script file]` hierarchy level. For more information, see "Store and Enable Junos Automation Scripts" on page 448.

NOTE: If the scripts are located in flash memory, you must configure the `load-scripts-from-flash` statement at the `[edit system scripts]` hierarchy level.

NOTE: For detailed information about the requirements and restrictions when executing Python automation scripts on devices running Junos OS, see "Requirements for Executing Python Automation Scripts on Junos Devices" on page 276.

You can configure the `arguments` statement to pass arguments to an event script as name/value pairs. The argument values can include variables that contain information about the triggering event or other received events. The event script can then reference this information during execution. For detailed information about passing arguments to event scripts, see "Configuring an Event Policy to Pass Arguments to an Event Script" on page 1031.

You can configure event scripts to write their output to a file. When the eventd process executes the scripts, it uploads the file with the script output to the location specified in the `destination` statement. In the `destination` statement, include a destination name that is configured at the `[edit event-options destinations]` hierarchy level. For more information, see "Example: Define Destinations for File Archiving by Event Policies" on page 1062.

In the `output-filename` statement, define a descriptive string that will be included in the filename. Each uploaded file also includes the hostname and timestamp in the filename to ensure that the each filename is unique. If a policy is triggered multiple times in a 1-second period, an index number is appended to the filename to ensure that the filenames are still unique. The index number range is 001 through 999.

Starting in Junos OS Release 14.1R3, the naming convention for a command output file generated by an event script is changed. When an event policy executes an event script and the script generates an output file, the name of the file depends on the version of Junos OS running on the device. Prior to Junos OS Release 14.1R3, the filename has the following naming convention:

```
hostname_output-filename_YYYYMMDD_HHMMSS_index-number
```

Starting in Junos OS Release 14.1R3, the filename places the ***output-filename*** string after the timestamp.

```
hostname_YYYYMMDD_HHMMSS_output-filename_index-number
```

For example, on a device named r1 running Junos OS Release 14.1R3 or a later release, if you configure the `output-filename` statement as **ifl-events**, and this event policy is triggered three times in 1 second, the files are named:

- **r1_20060623_132333_ifl-events**
- **r1_20060623_132333_ifl-events_001**
- **r1_20060623_132333_ifl-events_002**

For the `destination` and `output-filename` statements, there are four configuration scenarios:

- You can omit the `destination` and `output-filename` 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 both the `destination` and `output-filename` 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 file, and the file is sent to the destination specified in the configuration.

- You can include the `destination` statement in the configuration. You can 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 "$ABC123";   ## SECRET-DATA***
          }
      }
  }
  ```

  In this example, the **/var/cmd.txt** file resides on device2 . The event policy uses the File Transfer Protocol (FTP) to upload this file to the **/tmp** directory on 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 device1. The event policy renames the **/var/cmd.txt** file as **device2_*YYYYMMDD_HHMMSS*_cmd.txt_*index-number***.

- You can include the `destination` and `output-filename` 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.

The default and only format for event script output is Junos Extensible Markup Language (XML).

> **NOTE**: Event script output is always emitted in XML format even if you configure the `output-format text` statement under the `[edit event-options policy` *`policy-name`* `then event-script` *`filename`*`]` hierarchy.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
14.1R3	Starting in Junos OS Release 14.1R3, the naming convention for a command output file generated by an event script is changed.

RELATED DOCUMENTATION

Configuring an Event Policy to Pass Arguments to an Event Script | 1031

Change the Configuration Using an Event Script

IN THIS SECTION

You can configure an event policy to change the configuration in response to an event. Event policies can modify the configuration by invoking an event script that changes and commits the configuration or by using the `change-configuration` statement to execute configuration mode commands that change the configuration. Event scripts provide more flexibility than the `change-configuration` statement when modifying the configuration. For example, event scripts enable you to check for specific conditions, provide the configuration data in different formats, and specify how to merge the data with the existing configuration. In certain cases, for example on dual-Routing Engine devices that have nonstop active routing (NSR) enabled, event policies can only use event scripts to modify the configuration.

The following sections discuss using event scripts to modify the configuration.

How to Change the Configuration Using a SLAX or XSLT Event Script

SLAX and XSLT event scripts can call the `jcs:load-configuration` template to make structured changes to the Junos OS configuration. You must establish a connection with the target device before invoking the template to modify the configuration. For information about the template, see "jcs:load-configuration" on page 442 and "Change the Configuration Using SLAX and XSLT Scripts" on page 857.

The following SLAX event script opens a connection to the local device, calls the `jcs:load-configuration` template to modify and commit the configuration, and then closes the connection. All of the values required for the `jcs:load-configuration` template are defined as variables, which are then passed into the template as arguments.

```
version 1.2;
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> {

    /* Open a connection to the local device */
    var $connection = jcs:open();

    /* Define configuration change */
    var $configuration-change = <configuration> {
        <routing-options> {
            <static> {
                <route> {
                   <name>"198.51.100.0/24";
                    <next-hop>"10.1.3.1";
                }
```

```
            }
        }
    }

    /* Load and commit the configuration */
    var $load-action = "merge";
    var $options := {
        <commit-options> {
            <log> "Configuration modified through event script";
        }
    }
    var $results := { call jcs:load-configuration($connection, $action=$load-action,
$configuration=$configuration-change, $commit-options=$options); }

    /* Close the connection */
    var $close-results = jcs:close($connection);
}
}
```

To configure an event policy that invokes the SLAX event script for a given event:

1. Copy the script into the **/var/db/scripts/event** directory on the device, and provide a filename that uses the **.slax** extension, for example, **config-change.slax**.
2. Configure the event script.

   ```
   [edit]
   user@host# set event-options event-script file config-change.slax
   ```

3. Configure an event policy that invokes the event script for your specific event, for example:

   ```
   [edit]
   user@host# set event-options policy config-change events MY_EVENT
   user@host# set event-options policy config-change then event-script config-change.slax
   ```

4. Commit the configuration.

   ```
   [edit]
   user@host# commit
   ```

How to Change the Configuration Using a Python Event Script

Python scripts can use the Junos PyEZ library to make changes to the configuration on devices running Junos OS. The Junos PyEZ `jnpr.junos.utils.config.Config` utility provides instance methods to lock, load, commit, and unlock the configuration.

The following Python event script connects to the local device and updates and commits the configuration.

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
import jcs

with Device() as dev:
    with Config(dev) as cu:
        cu.load("set routing-options static route 198.51.100.0/24 next-hop 10.1.3.1",
format="set")
        cu.commit(comment="Configuration modified through event script", timeout=300)
```

To configure an event policy that invokes the Python event script for a given event:

1. Copy the script into the **/var/db/scripts/event** directory on the device, and provide a filename that uses the **.py** extension, for example, **config-change.py**.

   > **NOTE**: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

2. Enable the execution of unsigned Python scripts on the device.

   ```
   [edit]
   user@host# set system scripts language python3
   ```

3. Configure the event script and the user under whose permissions the script is executed.

   ```
   [edit]
   user@host# set event-options event-script file config-change.py python-script-user admin
   ```

4. Configure an event policy that invokes the event script for your specific event, for example:

```
[edit]
user@host# set event-options policy config-change events MY_EVENT
user@host# set event-options policy config-change then event-script config-change.py
```

5. Commit the configuration.

```
[edit]
user@host# commit
```

For more information about using Junos PyEZ to configure devices running Junos OS, see the Junos PyEZ Developer Guide.

How to Change the Configuration Using Event Scripts on Devices that have Nonstop Active Routing Enabled

When you use an event policy to change the configuration on a dual Routing Engine device that has nonstop active routing (NSR) enabled, we recommend that the event policy invoke an event script that commits the updated configuration on only the primary Routing Engine. This helps ensure that the update to the configuration and the subsequent commit operation are successful on both Routing Engines. The configuration is automatically synchronized to the backup Routing Engine, because the `commit synchronize` statement is configured at the `[edit system]` hierarchy level as part of the NSR configuration. Alternatively, if you use the `change-configuration` statement to modify the configuration, or if the event script does not commit the change on only the primary Routing Engine, both Routing Engines might simultaneously attempt to acquire a lock on the configuration database, thereby causing one or both commits to fail.

To create an event script that only configures and commits the configuration on the primary Routing Engine, include logic that tests whether the current Routing Engine is the primary Routing Engine. If the current Routing Engine is the primary Routing Engine, update the configuration and commit it.

The following SLAX event script connects to the local device and checks whether the current Routing Engine is the primary Routing Engine. If it is the primary Routing Engine, the script updates the configuration and then commits it.

```
version 1.2;
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> {

    /* Retrieve chassis information */
    var $rpc = <get-chassis-inventory>;
    var $chassis_rpc = jcs:invoke($rpc);
    var $current_state = $chassis_rpc/chassis/name;

    /* Open a connection to the local device */
    var $connection = jcs:open();

    /* Define configuration change */
    var $configuration-change =  <configuration> {
        <routing-options> {
            <static> {
                <route> {
                    <name>"198.51.100.0/24";
                    <next-hop>"10.1.3.1";
                }
            }
        }
    }

    /* Load and commit the configuration */
    var $load-action = "merge";
    var $options := {
        <commit-options> {
            <log> "Configuration modified through event script";
        }
    }
    if ($current_state == "Chassis") {
        var $results := { call jcs:load-configuration($connection, $action=$load-action,
$configuration = $configuration-change, $commit-options=$options); }
    }

    /* Close the connection */
    var $close-results = jcs:close($connection);
}
}
```

Similarly, the following Python event script connects to the local device and only updates and commits the configuration if the current Routing Engine is the primary Routing Engine:

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
import jcs

with Device() as dev:
    if("master" in dev.facts["current_re"]):
        with Config(dev) as cu:
            cu.load("set routing-options static route 198.51.100.0/24 next-hop 10.1.3.1",
format="set")
            cu.commit(comment="Configuration modified through event script", timeout=300)
```

RELATED DOCUMENTATION

Change the Configuration Using SLAX and XSLT Scripts | 857
Configure an Event Policy to Change the Configuration | 999

Configuring an Event Policy to Pass Arguments to an Event Script

IN THIS SECTION

When an event policy invokes an event script, the policy can pass arguments to the script. The following sections outline how to configure the arguments in the event policy and use the arguments within the event script:

Configuring Event Script Arguments in an Event Policy

You configure the arguments that an event policy passes to an event script within the `then` clause of the policy under the `event-script` *`filename`* `arguments` hierarchy. You can configure any number of arguments for each invoked event script.

```
[edit event-options policy policy-name then]
event-script filename {
    arguments {
        argument-name argument-value;
    }
}
```

You include arguments to the script as name/value pairs. The argument values can include variables that contain information about the triggering event or other received events. The event script can then reference this information during execution. 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 resolves to the value of the attribute associated with the triggering event. For example, `{$$.interface-name}` resolves to the interface name associated with the triggering event.

- `{$`*`event`*`.`*`attribute-name`*`}`—The single dollar sign with the event name (`$`*`event`*) notation represents the most recent event that matches *event*. When combined with an attribute name, the variable resolves to the value of the attribute associated with that event. For example, `{$COSD_CHAS_SCHED_MAP_INVALID.interface-name}` resolves to the interface name associated with 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
```

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>
...
```

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
  ...
```

Using Arguments in an Event Script

When an event policy invokes an event script, the event script can reference any of the arguments that are passed in by the policy. The names of the arguments in the event script must match the names of the arguments configured for that event script under the `[edit event-options policy` *`policy-name`* `then event-scripts` *`filename`* `arguments]` hierarchy in the configuration.

To use the arguments within a SLAX or XSLT event script, you must include a parameter declaration for each argument. The event script assigns the value for each script argument to the corresponding parameter of the same name, which can then be referenced throughout the script.

XSLT Syntax

```
<xsl:param name="argument-name"/>
```

SLAX Syntax

```
param $argument-name;
```

To use the arguments within a Python event script, you can use any valid means in the Python language. The following example uses the Python `argparse` module to process the script arguments. A `parser.add_argument` statement must be included for each argument passed to the script.

Python Syntax

```
import argparse

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-argument-name', required=True)
    args = parser.parse_args()

    # to use the argument reference args.argument-name

if __name__ == '__main__':
    main()
```

SEE ALSO

Execute Event Scripts in an Event Policy | 1021

arguments (Event Policy)

Configure 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 "Use Nonstandard System Log Messages to Trigger Event Policies" on page 986.

In addition, you can reference internally generated events, which are discussed in "Generate Internal Events to Trigger Event Policies" on page 980.

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.

Example: Ignore 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 ...
    }
}
```

Overview of Using 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. If one or more of the listed events occur, the system log message for the event is converted into a trap. 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 helpful if you use NMS traps rather than system log messages to monitor your network.

To configure an event 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 Juniper Networks enterprise-specific System Log MIB, whose object identifier is {jnxMibs 35}, provides support for this feature.

RELATED DOCUMENTATION

Example: Raise an SNMP Trap in Response to an Event | 1037

Example: Raise an SNMP Trap in Response to an Event

IN THIS SECTION

This example configures an event policy to raise a trap and to execute an event script in response to an event:

Requirements

A device running Junos OS, which is configured for SNMP.

Overview

The following example configures the event policy `raise-trap-on-ospf-nbrdown` to trigger on the `RPD_OSPF_NBRDOWN` event, which indicates a terminated OSPF adjacency with a neighboring router. The event policy action raises a trap in response to the event. The device sends a notification to the SNMP manager, if one is configured under the `[edit snmp]` hierarchy level.

Additionally, the event policy executes the event script **ospf.xsl** in response to this event and provides the affected interface as an argument to the script. The `$$rpd_ospf_nbrdown.interface-name` argument resolves to the interface name associated with the triggering event.

The event script output is recorded in the file **ospf-out**, and the output file is uploaded to the destination `mgmt-archives`, which is configured at the `[edit event-options destinations]` hierarchy level. To invoke an event script in an event policy, the event script must be present in the **/var/db/scripts/event** directory on the hard disk, and it must be enabled in the configuration.

Configuration

IN THIS SECTION

CLI Quick Configuration

To quickly configure this example, copy the following commands, paste them in a text file, remove any line breaks, change any details necessary to match your network configuration, and then copy and paste the commands into the CLI at the `[edit]` hierarchy level.

```
set event-options policy raise-trap-on-ospf-nbrdown events rpd_ospf_nbrdown
set event-options policy raise-trap-on-ospf-nbrdown then raise-trap
set event-options policy raise-trap-on-ospf-nbrdown then event-script ospf.xsl arguments
interface "{$$rpd_ospf_nbrdown.interface-name}"
set event-options policy raise-trap-on-ospf-nbrdown then event-script ospf.xsl output-filename
```

```
ospf-out
set event-options policy raise-trap-on-ospf-nbrdown then event-script ospf.xsl destination mgmt-
archives
```

Configuring the Event Policy

Step-by-Step Procedure

To configure an event policy that raises a trap on receipt of an event and optionally executes an event script:

1. Create and name the event-policy.

   ```
   [edit]
   user@R1# edit event-options policy raise-trap-on-ospf-nbrdown
   ```

2. Configure the event policy to match on the desired event, which in this example is the `RPD_OSPF_NBRDOWN` event.

   ```
   [edit event-options policy raise-trap-on-ospf-nbrdown]
   user@R1# set events rpd_ospf_nbrdown
   ```

3. Configure the event policy action to raise an SNMP trap in response to the event.

   ```
   [edit event-options policy raise-trap-on-ospf-nbrdown]
   user@R1# set then raise-trap
   ```

4. (Optional) Configure additional actions to take in response to the event.

   This example executes an event script and uploads the associated output file to a predefined destination.

   ```
   [edit event-options policy raise-trap-on-ospf-nbrdown]
   user@R1# set then event-script ospf.xsl arguments interface {$$rpd_ospf_nbrdown.interface-
   name}
   user@R1# set then event-script ospf.xsl output-filename ospf-out destination mgmt-archives
   ```

5. Commit the configuration.

```
user@R1# commit
```

Results

```
[edit event-options]
policy raise-trap-on-ospf-nbrdown {
    events rpd_ospf_nbrdown;
    then {
        event-script ospf.xsl {
            arguments {
                interface "{$$rpd_ospf_nbrdown.interface-name}";
            }
            output-filename ospf-out;
            destination mgmt-archives;
        }
        raise-trap;
    }
}
```

RELATED DOCUMENTATION

Overview of Using Event Policies to Raise SNMP Traps | 1037

Understanding the Event System Log Priority in an Event Policy

Starting in Junos OS Release 12.1, you can configure an event policy to override the default system log priority of a triggering event so that the system logs the event with a different facility type, severity level, or both. To override the priority of the triggering event, configure the `priority-override` statement at the `[edit event-options policy *policy-name* then]` hierarchy level. To override the facility type with which the triggering event is logged, include the `facility` statement and the new facility type. To override the severity level with which the triggering event is logged, include the `severity` statement and the new severity level.

Junos OS processes generate system log messages, or event notifications, to record the events that occur on a routing, switching, or security platform. Each system log message identifies the Junos OS process that generated the message and describes the operation or error that occurred. The Junos OS event process (eventd) receives the event notifications, and configured event policies instruct the eventd process to perform a set of actions upon receipt of specific events or correlated events.

Each system log message belongs to a facility, which groups messages that either are generated by the same source (such as a software process) or concern a similar condition or activity (such as authentication attempts). Each message is also preassigned a severity level, which indicates how seriously the triggering event affects the functions of the routing, switching, or security platform. A message's facility and severity level are together referred to as its priority. For more information about facility and severity levels, see Junos OS System Logging Facilities and Message Severity Levels.

When you configure logging on a device for a specific facility and destination, you also specify a severity level. Messages from that facility that are rated at the configured severity level or higher are logged. To log related events with different severity levels in the same log file, you must filter events using the lowest severity level of any of the events from that facility to be logged. This can result in unwieldy log files that are difficult and time-consuming to parse.

For example, Junos OS logs the protocol UP and DOWN events with different severity levels. Both the SNMP_TRAP_LINK_DOWN and SNMP_TRAP_LINK_UP events have a facility of 'daemon', but the SNMP_TRAP_LINK_DOWN event has a severity level of 'warning', and the SNMP_TRAP_LINK_UP event has a severity level of 'info'. Normally, when you configure a system log file, you must filter events to that file using the lower severity level of 'info' in order to log both of the events.

The event policy `priority-override` statement enables you to customize the priority of the triggering event so that it is logged using a different facility type and severity level. Suppose you configure a system log file to filter events of facility 'daemon' and severity 'notice', and you have event policies that trigger on the RPD_ISIS_ADJDOWN and RPD_ISIS_ADJUP events. When the system generates an RPD_ISIS_ADJDOWN message reporting that the IS-IS adjacency with a neighboring router was terminated, this message is logged. However, if the system subsequently generates an RPD_ISIS_ADJUP event notification reporting that the IS-IS adjacency has been restored, by default, the message is not logged, because it has a lower severity level of 'info'. In the event policy that triggers on the RPD_ISIS_ADJUP event, you can configure the associated priority so that the triggering RPD_ISIS_ADJUP event is logged with a severity level of 'notice' and is captured in the configured log file.

> **NOTE**: Event policies are executed in the order in which they appear in the configuration. When you configure multiple event policies to override the priority of the same event, the event is logged based on the priority set by the last executed event policy to change it.

RELATED DOCUMENTATION

Example: Configuring the Event System Log Priority in an Event Policy

IN THIS SECTION

It is necessary to log events when monitoring, managing, and troubleshooting routing, switching, and security devices. You can configure an event policy to override the priority of its triggering event so that it is logged based on a different facility type and severity level. This enables the event to be logged even if the system filters events to the destination log file using a different facility type or a higher severity level.

This example simulates an SNMP_TRAP_LINK_UP event for a specific interface. Upon receipt of the event, the event policy overrides the severity level of the event so that it is captured in the configured log file.

Requirements

- Routing, switching, or security device running Junos OS Release 12.1 or later.
- Interface is configured and active. This example uses the ge-0/3/1.0 interface.

Overview

This example configures two log files to capture events of facility 'daemon'. One log file is configured to filter for events of severity 'warning' or higher, and the second log file is configured to filter for events of severity 'info' or higher.

The configured event policy triggers on the SNMP_TRAP_LINK_UP event for interface ge-0/3/1.0. The example generates an SNMP_TRAP_LINK_DOWN event followed by an SNMP_TRAP_LINK_UP event for the ge-0/3/1.0 interface. The SNMP_TRAP_LINK_DOWN event, which has a severity level of 'warning' is captured in both configured log files. Upon receipt of the SNMP_TRAP_LINK_UP event, the event policy overrides the severity level of the event to 'warning' so that it is also captured in the log file that filters for events of severity 'warning'. By default, if the event policy does not override the severity level of this event, it is only captured in the log file that filters for the severity level 'info'.

Configuration

IN THIS SECTION

CLI Quick Configuration

To quickly configure this example, copy the following commands, paste them in a text file, remove any line breaks, change any details necessary to match your network configuration, and then copy and paste the commands into the CLI at the `[edit]` hierarchy level:

```
set system syslog file syslog-event-daemon-info daemon info
set system syslog file syslog-event-daemon-warning daemon warning
set event-options policy log-on-snmp-trap-link-up events snmp_trap_link_up
set event-options policy log-on-snmp-trap-link-up attributes-match snmp_trap_link_up.interface-
name matches ge-0/3/1.0
set event-options policy log-on-snmp-trap-link-up then priority-override severity warning
```

Configuring the Log Files

Step-by-Step Procedure

1. Configure two log files at the `[edit system syslog]` hierarchy level to record events of facility `daemon`.

   Configure one log to record events of severity 'info' or higher and one log file to record events of severity 'warning' or higher.

   ```
   [edit system syslog]
   bsmith@R1# set file syslog-event-daemon-info daemon info
   bsmith@R1# set file syslog-event-daemon-warning daemon warning
   ```

2. Commit the configuration.

   ```
   bsmith@R1# commit
   ```

3. To manually test the logging of the events, take the ge-0/3/1.0 logical interface temporarily offline, and then bring it back up.

   This generates an SNMP_TRAP_LINK_DOWN event followed by an SNMP_TRAP_LINK_UP event.

   ```
   [edit]
   bsmith@R1# set interfaces ge-0/3/1 unit 0 disable
   bsmith@R1# commit
   bsmith@R1# delete interfaces ge-0/3/1 unit 0 disable
   bsmith@R1# commit
   ```

Results

```
[edit]
system {
    syslog {
        file syslog-event-daemon-info {
            daemon info;
        }
        file syslog-event-daemon-warning {
            daemon warning;
        }
```

```
    }
}
```

Verifying the Default System Log Priority of the Events

Purpose

Verify that the system generated the SNMP_TRAP_LINK_DOWN and SNMP_TRAP_LINK_UP events for the ge-0/3/1.0 interface, and note where each event is logged.

Action

Review the contents of the **syslog-event-daemon-info** file configured in Step "1" on page 1044 of the previous procedure. The output shows that the ge-0/3/1.0 interface was brought down and back up and generated an SNMP_TRAP_LINK_DOWN event followed by an SNMP_TRAP_LINK_UP event.

```
bsmith@R1> show log syslog-event-daemon-info
Oct 24 13:22:17  R1 mib2d[1394]: SNMP_TRAP_LINK_DOWN: ifIndex 539, ifAdminStatus down(2),
ifOperStatus down(2), ifName ge-0/3/1.0
...
Oct 24 13:22:29  R1 mib2d[1394]: SNMP_TRAP_LINK_UP: ifIndex 539, ifAdminStatus up(1),
ifOperStatus up(1), ifName ge-0/3/1.0
```

Review the contents of the **syslog-event-daemon-warning** file configured in Step "1" on page 1044 of the previous procedure. Because the severity level of the SNMP_TRAP_LINK_UP event is 'info', it does not appear in a log file that is configured to only record events of severity 'warning' or higher. By default, this system log file captures the SNMP_TRAP_LINK_DOWN events, but does not capture the SNMP_TRAP_LINK_UP events.

```
bsmith@R1> show log syslog-event-daemon-warning
Oct 24 13:22:17  R1 mib2d[1394]: SNMP_TRAP_LINK_DOWN: ifIndex 539, ifAdminStatus down(2),
ifOperStatus down(2), ifName ge-0/3/1.0
```

Meaning

Because the SNMP_TRAP_LINK_UP event has a default severity of 'info', it is not forwarded to log files that are configured to capture events of higher severity.

Configuring the Event Policy

Step-by-Step Procedure

1. Create and name the event-policy.

```
[edit]
bsmith@R1# edit event-options policy log-on-snmp-trap-link-up
```

2. Configure the `events` statement.

   For this example, the event policy triggers on the SNMP_TRAP_LINK_UP event. Set the `attributes-match` statement so that the policy triggers only if the SNMP_TRAP_LINK_UP event occurs for the ge-0/3/1.0 interface.

```
[edit event-options policy log-on-snmp-trap-link-up]
bsmith@R1# set events snmp_trap_link_up
bsmith@R1# set attributes-match snmp_trap_link_up.interface-name matches ge-0/3/1.0
```

3. Configure the `priority-override` event policy action, and include the `severity` statement with a value of `warning`.

```
[edit event-options policy log-on-snmp-trap-link-up]
bsmith@R1# set then priority-override severity warning
```

4. Commit the configuration.

```
bsmith@R1# commit
```

5. To manually test the event policy, take the ge-0/3/1.0 logical interface temporarily offline, and then bring it back up. This generates an SNMP_TRAP_LINK_DOWN event followed by an SNMP_TRAP_LINK_UP event.

```
[edit]
bsmith@R1# set interfaces ge-0/3/1 unit 0 disable
bsmith@R1# commit
bsmith@R1# delete interfaces ge-0/3/1 unit 0 disable
bsmith@R1# commit
```

Results

```
[edit]
event-options {
    policy log-on-snmp-trap-link-up {
        events snmp_trap_link_up;
        attributes-match {
            snmp_trap_link_up.interface-name matches ge-0/3/1.0;
        }
        then {
            priority-override {
                severity warning;
            }
        }
    }
}
```

Verification

IN THIS SECTION

Confirm that the configuration is working properly.

Verifying the Configured System Log Priority of the Events

Purpose

Verify that the system generated the SNMP_TRAP_LINK_DOWN and SNMP_TRAP_LINK_UP events for the ge-0/3/1.0 interface, and note where each event is logged.

Action

Review the contents of the **syslog-event-daemon-warning** file. Because the event policy overrides the severity level of the SNMP_TRAP_LINK_UP event, it now appears in the log file that is configured to only record events of severity 'warning' or higher. By default, this system log file captures the SNMP_TRAP_LINK_DOWN events, but does not capture the SNMP_TRAP_LINK_UP events.

```
bsmith@R1> show log syslog-event-daemon-warning
Oct 24 13:29:48  R1 mib2d[1394]: SNMP_TRAP_LINK_DOWN: ifIndex 539, ifAdminStatus down(2),
ifOperStatus down(2), ifName ge-0/3/1.0
Oct 24 13:30:02  R1 mib2d[1394]: SNMP_TRAP_LINK_UP: ifIndex 539, ifAdminStatus up(1),
ifOperStatus up(1), ifName ge-0/3/1.0
```

Meaning

Although the SNMP_TRAP_LINK_UP event has a severity of 'info', configuring the `priority-override` statement with a severity of 'warning' causes the event to be forwarded to the system logs with the configured severity level. The event can be captured in logs that filter for a different facility type and a higher severity level.

RELATED DOCUMENTATION

Understanding the Event System Log Priority in an Event Policy | **1040**

Junos OS System Logging Facilities and Message Severity Levels

Specifying the Facility and Severity of Messages to Include in the Log

facility

priority-override (Event Policy)

severity (Event Policy)

Example: Limit 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> {
            <name> "event-detail";
            <events> "ui_login_event";
            <then> {
                <event-script> {
                    <name> "event_detail.slax";
                    <output-filename> "foo";
                    <destination> {
                        <name> "foo";
                    }
                }
            }
         }
    }
}

match / {
    <event-script-results> {
        <event-triggered-this-policy> {
            expr event-script-input/trigger-event/id;
        }
        <type-of-event> {
            expr event-script-input/trigger-event/type;
        }
        <process-name> {
```

```
            expr event-script-input/trigger-event/attribute-list/attribute/name;
        }
    }
}
```

CHAPTER 37

Configure Event Policy File Archiving

IN THIS CHAPTER

Event Policy File Archiving

SUMMARY

Configure event policies to upload relevant files to predefined archive sites.

IN THIS SECTION

Event Policy File Archiving Overview

Various types of files are useful in diagnosing events and troubleshooting network issues. When an event policy action generates output files, you can archive the files for later analysis. Similarly, you might want to archive system files, including system log files, core files, and configuration files, from the time an event occurs.

When an event occurs, you can upload relevant files to a specified location for analysis. To archive files from an event policy, configure one or more *destinations* specifying the archive sites to which the files are uploaded. You then reference the configured destinations within event policies. When the event policy triggers, it uploads the files to the specified archive site.

You can configure a transfer delay for event policy archive operations. A transfer delay enables you to specify the number of seconds the event process (eventd) waits before uploading one or more files. A transfer delay helps ensure that a large file, such as a core file, is completely generated before the upload begins.

You can associate transfer delays with a destination and with an event policy action. If you associate a transfer delay with a destination, the transfer delay applies to all file upload actions that use that destination. You can also assign a transfer delay to an event policy action. For example, you might have multiple event policy actions that use the same destination, and for some of these event policy actions, you want a transfer delay, and for other event policy actions you want no transfer delay. If you configure a transfer delay for a destination, and you also configure a transfer delay for the event policy action, the resulting transfer delay is the sum of the two delays.

Transient network problems can cause a file upload operation to fail. If the upload fails for any reason, by default, the event policy does not retry the upload. However, you can configure an event policy to retry the file upload operation a specified number of times if the initial upload fails. You can also configure the time interval between each retry attempt.

Define Destinations for Event Policy File Archiving

When an event occurs, you can upload relevant files to a specified location for analysis. To archive files from an event policy, you must first configure one or more *destinations* specifying the archive sites to which the files are uploaded. You then reference the configured destinations within event policies.

To define a destination archive site, 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;
    }
}
```

For each destination, configure one or more archive site URIs, which are the actual sites to which the files are uploaded. If you specify multiple archive site URIs for a given destination, the device attempts to transfer the files to the first archive site in the list and only uses subsequent archive sites if the transfer to the previous site fails. If an archive site requires authentication to log in, you can configure a plain-text password for that site. The device stores the password as an encrypted value in the configuration database.

The archive site URI is a file URI, an active or passive FTP URI, a Secure FTP (SFTP) URI, or a Secure Copy (SCP) URI. Local device directories are also supported (for example, **/var/tmp**). When you specify the archive site URI, do not add a forward slash (/) to the end of the URI.

- file:<//host>/path
- ftp://username@host:<port>url-path
- pasvftp://username@host:<port>url-path
- sftp://username@host:<port>url-path
- scp://username@host:<port>url-path
- <path>/<filename>

You can also define a transfer delay for each destination. The transfer delay is the number of seconds the event process (eventd) waits before uploading one or more files to that destination. A transfer delay helps to ensure that a large file, such as a core file, is completely generated before the upload begins.

To define a destination archive site to which event policies can upload files:

1. Define the destination name, which is a user-defined identifier that is referenced by event policies.

   You can define multiple destinations with one or more archive sites.

   ```
   [edit event-options]
   user@host# edit destinations destination-name
   ```

   For example:

   ```
   [edit event-options]
   user@host# edit destinations mgmt-archives
   ```

2. Configure one or more archive site URIs.

If an archive site requires authentication, configure the required plain-text password for that site.

```
[edit event-options destinations destination-name]
user@host# set archive-sites URI1 password password
user@host# set archive-sites URI2
```

For example:

```
[edit event-options destinations mgmt-archives]
user@host# set archive-sites "scp://username@example.com/test" password PaSsWoRd
user@host# set archive-sites /var/log
```

3. (Optional) Configure the transfer delay, in seconds, associated with the destination.

```
[edit event-options destinations destination-name]
user@host# set transfer-delay seconds
```

The following example configures a delay of five seconds for files uploaded to the mgmt-archives destination.

```
[edit event-options destinations mgmt-archives]
user@host# set transfer-delay 5
```

Configure an Event Policy to Upload Files

You can configure an event policy to upload existing system files or to upload the output files generated from an invoked event-script or operational command at the time an event occurs. The device uploads the files to the location referenced in the `destination` statement configured for that event policy action. You must specify a destination name that is configured at the `[edit event-options destinations]` hierarchy level.

The following examples configure various event policy actions to upload specific files to an existing destination. For each event policy, you must also configure the appropriate events and include any other required statements.

Upload System Files

To configure an event policy to upload system files to a configured destination:

1. Configure the `upload` event policy action, and specify the files to upload and the destination site.

You can include multiple `upload` statements, and the `filename` statement can use *filename globbing* to specify multiple files.

```
[edit event-options policy policy-name then]
user@host# set upload filename (filename | committed) destination destination-name
```

The following event policy action uploads the committed configuration file and also uploads all files that are located in the **/var/log** directory and start with the string "messages".

```
[edit event-options policy policy1 then]
user@host# set upload filename committed destination mgmt-archives
user@host# set upload filename /var/log/messages* destination mgmt-archives
```

2. (Optional) Configure a transfer delay or the retry option as described in "Configure a Delay Before Files Are Uploaded" on page 1057 and "Configure an Event Policy to Retry the File Upload Action" on page 1060

Upload Command Output Files

An event policy can include the `execute-commands` event policy action to execute commands in response to an event and write the command output to a file. To configure an event policy to upload the command output file to a configured destination:

1. In the `execute-commands` event policy action, configure a destination.

```
[edit event-options policy policy-name then]
user@host# set execute-commands destination destination-name
```

For example:

```
[edit event-options policy policy1 then]
user@host# set execute-commands destination mgmt-archives
```

2. Define a descriptive string that will be included in the filename of the output file.

```
[edit event-options policy policy-name then]
user@host# set execute-commands output-filename string
```

For example:

```
[edit event-options policy policy1 then]
user@host# set execute-commands output-filename intf-info
```

3. (Optional) Configure a transfer delay or the retry option as described in "Configure a Delay Before Files Are Uploaded" on page 1057 and "Configure an Event Policy to Retry the File Upload Action" on page 1060

Upload Event Script Output Files

When an event policy executes an event script in response to an event, the event script can write output to a file. To configure an event policy to upload the generated output file to a configured destination:

1. In the `event-script` event policy action, configure a destination.

```
[edit event-options policy policy-name then]
user@host# set event-script filename destination destination-name
```

For example:

```
[edit event-options policy policy1 then]
user@host# set event-script get-intf-info destination mgmt-archives
```

2. Define a descriptive string that will be included in the filename of the output file.

```
[edit event-options policy policy-name then]
user@host# set event-script filename output-filename string
```

For example:

```
[edit event-options policy policy1 then]
user@host# set event-script get-intf-info output-filename intf-info
```

3. (Optional) Configure a transfer delay or the retry option as described in "Configure a Delay Before Files Are Uploaded" on page 1057 and "Configure an Event Policy to Retry the File Upload Action" on page 1060

Filenames for Uploaded Files

When an event policy action uploads files, the filename for each uploaded file includes the hostname and timestamp to ensure that it is unique. The name of the file depends on the Junos OS software version. Starting in Junos OS Release 14.1R3, the filename has the following naming convention:

```
hostname_YYYYMMDD_HHMMSS_output-filename_index-number
```

In earlier releases, the filename has the following naming convention:

```
hostname_output-filename_YYYYMMDD_HHMMSS_index-number
```

The ***output-filename*** string is either the name of an existing file or the value of the `output-filename` statement configured for that event policy action. If a device triggers an event policy multiple times in a 1-second period, it appends an index number is to each filename to ensure that the filenames are still unique. The index number range is 001 through 999.

For example, suppose you have an event policy action with `output-filename` configured as `rpd-messages` on device r1 running Junos OS Release 21.1R1. If the event policy triggers 3 times in 1 second, the filenames would be similar to the following:

- **r1_20210623_132333_rpd-messages**
- **r1_20210623_132333_rpd-messages_001**
- **r1_20210623_132333_rpd-messages_002**

Configure a Delay Before Files Are Uploaded

You can configure an event policy to upload existing system files or to upload the output files generated from an invoked event-script or operational command at the time an event occurs. For event policy upload operations, you can configure a transfer delay to specify the number of seconds the event process (eventd) waits before uploading one or more files. By configuring a transfer delay, you can better ensure that a large file, such as a core file, is completely generated before the upload begins.

You can associate transfer delays with a destination and with an event policy action. If you associate a transfer delay with a destination, the transfer delay applies to all file upload actions that use that destination. You can also assign a transfer delay to an event policy action. For example, you might have multiple event policy actions that use the same destination, and for some of these event policy actions, you want a transfer delay, and for other event policy actions you want no transfer delay.

If you configure a transfer delay for a destination 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 delays.

```
Total transfer delay = transfer-delay (destination) + transfer-delay (event-policy-action)
```

To configure a transfer delay for a destination:

Configure the delay, in seconds, associated with the destination.

```
[edit event-options destinations destination-name]
user@host# set transfer-delay seconds
```

For example, the following configuration sets a transfer delay of five seconds for the mgmt-archives destination.

```
[edit event-options destinations mgmt-archives]
user@host# set transfer-delay 5
```

To configure a transfer delay for a specific event policy action:

1. In the appropriate event policy action hierarchy, configure the delay, in seconds.

    ```
    [edit event-options policy policy-name then]
    user@host# set event-script filename destination destination-name transfer-delay seconds
    ```

    ```
    [edit event-options policy policy-name then]
    user@host# set execute-commands destination destination-name transfer-delay seconds
    ```

    ```
    [edit event-options policy policy-name then]
    user@host# set upload filename (filename | committed) destination destination-name transfer-
    delay seconds
    ```

For example:

```
[edit event-options policy policy1 then]
user@host# set event-script get-intf-info.py destination mgmt-archives transfer-delay 5
```

```
[edit event-options policy policy2 then]
user@host# set execute-commands destination mgmt-archives transfer-delay 7
```

```
[edit event-options policy policy3 then]
user@host# set upload filename committed destination mgmt-archives transfer-delay 3
```

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;
        destination some-dest;
    }
}
policy policy2 {
    events e2;
    then {
        event-script bar.xsl {
            output-filename event-script-output;
            destination some-dest;
        }
    }
}
destinations {
    some-dest {
```

```
        transfer-delay 2;
        archive-sites {
            "scp://robot@my.big.com/foo/moo" password "$9$wisoGDjqfQnHqIclMN-HqmP5F"; ## SECRET-
DATA
            "scp://robot@my.little.com/foo/moo" password "$9$uova0RSrlMXNbKMDkPQ9CKM8Lxd"; ##
SECRET-DATA
        }
    }
}
```

Configure an Event Policy to Retry the File Upload Action

You can configure an event policy to upload existing system files or to upload the output files generated from an invoked event-script or operational command at the time an event occurs. Transient network problems can cause a file upload operation to fail. If the upload fails for any reason, by default, the event policy does not retry the upload. However, you can configure an event policy to retry a file upload operation.

You configure the file upload retry option for a given event policy action. To configure the retry option, include the `retry-count` and `retry-interval` statements:

```
retry-count number retry-interval seconds;
```

Where:

- `retry-count`—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.
- `retry-interval`—Number of seconds between each upload attempt.

To configure the event policy to retry a file upload operation for a given event policy action:

Include the `retry-count` and `retry-interval` statements for the event policy action's `destination` statement.

```
[edit event-options policy policy-name then]
user@host# set event-script filename destination destination-name retry-count number retry-
interval seconds
```

```
[edit event-options policy policy-name then]
user@host# set execute-commands destination destination-name retry-count number retry-interval
seconds
```

```
[edit event-options policy policy-name then]
user@host# set upload filename (filename | committed) destination destination-name retry-count
number retry-interval seconds
```

For example:

```
[edit event-options policy policy1 then]
user@host# set event-script get-intf-info.py destination mgmt-archives retry-count 5 retry-
interval 3
```

```
[edit event-options policy policy2 then]
user@host# set execute-commands destination mgmt-archives retry-count 10 retry-interval 4
```

```
[edit event-options policy policy3 then]
user@host# set upload filename committed destination mgmt-archives retry-count 2 retry-interval
10
```

RELATED DOCUMENTATION

Example: Define Destinations for File Archiving by Event Policies | **1062**
Example: Configuring an Event Policy to Upload Files | **1065**
Example: Configuring the Delay Before Files Are Uploaded by an Event Policy | **1072**

Example: Define Destinations for File Archiving by Event Policies

IN THIS SECTION

This example configures an archive site for event policies. Event policy actions that reference the configured destination upload specified files to that site.

Requirements

This example uses a Junos device. No additional configuration beyond device initialization is required before configuring this example.

Overview

When an event policy action generates output files, you can archive the files for later analysis. Similarly, you might want to archive system files, including system log files, core files, and configuration files, from the time an event occurs.

When an event occurs, you can upload relevant files to a specified location. To archive files, configure one or more *destinations* specifying the archive sites to which the files are uploaded. To upload the files when the device executes an event policy, you must reference the configured destinations within an event policy.

This example configures a new archive destination named mgmt-archives, which can be referenced in event policies for file archiving. The example configures two archive sites for this destination. The first site is the Secure Copy URI "scp://username@example.com/test" for which a password is configured. The second site is a directory on the local device. The device attempts to transfer to the first archive site in the list, moving to the next site only if the transfer to the first site fails. The example configures a transfer delay of five seconds for all files uploaded to the mgmt-archives archive site.

Configuration

IN THIS SECTION

Procedure

CLI Quick Configuration

To quickly configure this example, copy the following commands, paste them in a text file, remove any line breaks, change any details necessary to match your network configuration, and then copy and paste the commands into the CLI at the [edit] hierarchy level:

```
set event-options destinations mgmt-archives archive-sites "scp://username@example.com/test"
password PaSsWoRd
set event-options destinations mgmt-archives archive-sites /var/log
set event-options destinations mgmt-archives transfer-delay 5
```

Step-by-Step Procedure

Configure a new archive destination named mgmt-archives that can be referenced by event policies.

1. Configure the identifier and associated archive sites for each destination.

   The device transfers to the first archive site in the list, moving to the next site only if the transfer to the first site fails.

   ```
   [edit event-options destinations]
   user@host# set mgmt-archives archive-sites scp://username@example.com/test
   user@host# set mgmt-archives archive-sites /var/log
   ```

2. If an archive site requires authentication, configure the required plain-text password for that site.

   ```
   [edit event-options destinations]
   user@host# set mgmt-archives archive-sites scp://username@example.com/test password PaSsWoRd
   ```

3. (Optional) Configure the transfer delay, in seconds, associated with each destination. The mgmt-archives destination has a transfer delay of five seconds.

```
[edit event-options destinations]
user@host# set mgmt-archives transfer-delay 5
```

4. Commit the configuration.

```
user@host# commit
```

5. You can reference configured destinations in an event policy. For information about referencing destinations in event policies, see "Example: Configuring an Event Policy to Upload Files" on page 1065 and "Configure an Event Policy to Execute Operational Mode Commands" on page 995.

Verification

IN THIS SECTION

Verifying the Configuration

Purpose

Issue the `show configuration event-options` operational mode command to review the resulting configuration.

Action

```
user@host> show configuration event-options
destinations {
    mgmt-archives {
        transfer-delay 5;
        archive-sites {
            "scp://username@example.com/test" password "$ABC123"; ## SECRET-DATA
            /var/log;
        }
```

```
    }
}
```

Meaning

In the sample output, the mgmt-archives destination has two archive sites and a transfer delay of five seconds. You can now reference this destination in event policies. When you reference the mgmt-archives destination in an event policy, the device uploads the specified files to the first archive site after a five second delay. If the transfer to the first archive fails, the device attempts to upload the files to the **/var/log** archive site.

Note that although the plain-text password is visible when you configure it, the configuration displays the encrypted password.

RELATED DOCUMENTATION

Example: Configuring an Event Policy to Upload Files | 1065
destinations (Event Policy)

Example: Configuring an Event Policy to Upload Files

IN THIS SECTION

This example configures event policy actions that upload relevant files to a specified location for analysis.

Requirements

Before you begin:

- Configure the destinations that you will reference in the event policy. See "Example: Define Destinations for File Archiving by Event Policies" on page 1062.
- Configure the general event policy and triggering events.

Overview

When an event policy action generates output files, you can archive the files for later analysis. Similarly, you might want to archive system files, including system log files, core files, and configuration files, from the time an event occurs. You can configure an event policy to upload existing system files or to upload the output files generated from an invoked event-script or command at the time an event occurs. This section outlines the configuration required for uploading each of these output files using an event policy.

When you configure an event policy to upload files, the device uploads the relevant files to the location referenced in the `destination` statement configured for that event policy action. You must specify a destination name that is configured at the `[edit event-options destinations]` hierarchy level.

In this example, `policy1` consists of the following statements, where *e1* is the triggering event. The example then configures the event policy to upload a log file and the committed configuration file as well as the output files generated from the `execute-commands` and `event-script` actions.

```
[edit event-options policy policy1]
events e1;
then {
    execute-commands {
        commands {
            "show interfaces brief ge-*";
        }
    }
    event-script event-script1;
}
```

Configuration

IN THIS SECTION

CLI Quick Configuration

To quickly configure this example, copy the following commands, paste them in a text file, remove any line breaks, change any details necessary to match your network configuration, and then copy and paste the commands into the CLI at the `[edit]` hierarchy level:

```
set event-options policy policy1 then upload filename /var/log/messages destination mgmt-
archives transfer-delay 4
set event-options policy policy1 then upload filename /var/log/messages destination mgmt-
archives retry-count 5 retry-interval 4
set event-options policy policy1 then upload filename /var/log/messages destination mgmt-
archives user-name admin
set event-options policy policy1 then upload filename /var/log/messages destination mgmt-server
set event-options policy policy1 then upload filename committed destination mgmt-archives
set event-options policy policy1 then execute-commands output-filename ge-interfaces
set event-options policy policy1 then execute-commands destination mgmt-archives transfer-delay 5
set event-options policy policy1 then execute-commands destination mgmt-archives retry-count 5
retry-interval 4
set event-options policy policy1 then event-script event-script1 output-filename policy1-script-
output
set event-options policy policy1 then event-script event-script1 destination mgmt-archives
transfer-delay 5
set event-options policy policy1 then event-script event-script1 destination mgmt-archives retry-
count 5 retry-interval 4
```

Uploading System Files

Step-by-Step Procedure

Configure the event policy `policy1` to upload the system file **/var/log/messages** to the archive sites mgmt-archives and mgmt-server. Additionally, upload the committed configuration to the archive site mgmt-archives. The destination archive sites should already be configured at the `[edit event-options destinations]` hierarchy level

1. Configure the `upload` statement, and include the file to archive and the destination archive site.

```
[edit event-options policy policy1 then]
bsmith@R1# set upload filename /var/log/messages destination mgmt-archives
bsmith@R1# set upload filename /var/log/messages destination mgmt-server
```

2. To upload the committed configuration file, specify the filename value as **committed**.

```
[edit event-options policy policy1 then]
bsmith@R1# set upload filename committed destination mgmt-archives
```

3. (Optional) Configure the transfer delay associated with each file and destination.

   The following command configures a 4-second transfer delay when the **/var/log/messages** file is uploaded to the mgmt-archives destination. If you also configure a transfer delay for the destination, the total delay is the sum of the two delays.

```
[edit event-options policy policy1 then]
bsmith@R1# set upload filename /var/log/messages destination mgmt-archives transfer-delay 4
```

4. (Optional) Configure the retry count and retry interval associated with each file and destination.

   In this example, if the **/var/log/messages** file fails to upload to the mgmt-archives site, the system attempts the upload up to 5 more times and waits 4 seconds in between each attempt.

```
[edit event-options policy policy1 then]
bsmith@R1# set upload filename /var/log/messages destination mgmt-archives retry-count 5
retry-interval 4
```

5. (Optional) Configure the username for the file's upload operation. The system uploads the file using the privileges of the specified user.

```
[edit event-options policy policy1 then]
bsmith@R1# set upload filename /var/log/messages destination mgmt-archives user-name admin
```

6. Commit the configuration.

```
[edit event-options policy policy1 then]
bsmith@R1# commit
```

Uploading Command Output Files

Step-by-Step Procedure

When the event policy invokes the `execute-commands` action, the command output can be written to a file. Configure the event policy `policy1` to write command output to a file and upload the generated file to the destination mgmt-archives, which is already configured at the `[edit event-options destinations]` hierarchy level.

1. Define a descriptive string that will be included in the filename of the output file.

```
[edit event-options policy policy1 then]
bsmith@R1# set execute-commands output-filename ge-interfaces
```

2. Configure the `destination` statement to upload the generated file to the specified archive site.

```
[edit event-options policy policy1 then]
bsmith@R1# set execute-commands destination mgmt-archives
```

3. (Optional) Configure the transfer delay for the upload operation, which in this example is 5 seconds.

```
[edit event-options policy policy1 then]
bsmith@R1# set execute-commands destination mgmt-archives transfer-delay 5
```

4. (Optional) Configure the retry count and retry interval for the upload operation.

   In this example, if the upload operation fails, the system attempts the upload up to 5 more times and waits 4 seconds in between each attempt.

```
[edit event-options policy policy1 then]
bsmith@R1# set execute-commands destination mgmt-archives retry-count 5 retry-interval 4
```

5. Commit the configuration.

```
[edit event-options policy policy1 then]
bsmith@R1# commit
```

Uploading Event Script Output Files

Step-by-Step Procedure

When the event policy invokes an event script, the script output can be written to a file. Configure the event policy `policy1` to write event-script output to a file and upload the generated file to the destination mgmt-archives, which is already configured at the `[edit event-options destinations]` hierarchy level. In this example, the event policy invokes an event script named **event-script1**.

1. Define a descriptive string that will be included in the filename of the output file.

```
[edit event-options policy policy1 then]
bsmith@R1# set event-script event-script1 output-filename policy1-script-output
```

2. Configure the `destination` statement to upload the generated file to the desired archive site.

```
[edit event-options policy policy1 then]
bsmith@R1# set event-script event-script1 destination mgmt-archives
```

3. (Optional) Configure the transfer delay for the upload operation, which in this example is 5 seconds.

```
[edit event-options policy policy1 then]
bsmith@R1# set event-script event-script1 destination mgmt-archives transfer-delay 5
```

4. (Optional) Configure the retry count and retry interval for the upload operation.

   In this example, if the upload fails, the system attempts the upload up to 5 more times and waits 4 seconds in between each attempt.

```
[edit event-options policy policy1 then]
bsmith@R1# set event-script event-script1 destination mgmt-archives retry-count 5 retry-
interval 4
```

5. Commit the configuration.

```
[edit event-options policy policy1 then]
bsmith@R1# commit
```

Results

```
[edit event-options policy policy1 then]
upload filename /var/log/messages destination mgmt-archives {
    user-name admin;
    transfer-delay 4;
    retry-count 5 retry-interval 4;
}
upload filename /var/log/messages destination mgmt-server
upload filename committed destination mgmt-archives;
execute-commands {
    commands {
        "show interfaces brief ge-*";
    }
    output-filename ge-interfaces;
    destination mgmt-archives {
        transfer-delay 5;
        retry-count 5 retry-interval 4;
    }
}
event-script event-script1 {
    output-filename policy1-script-output;
    destination mgmt-archives {
        transfer-delay 5;
        retry-count 5 retry-interval 4;
    }
}
```

Verification

IN THIS SECTION

Verifying the Upload

Purpose

When the configured event triggers the event policy, the system uploads the generated output files and the specified system files to the URL defined in the mgmt-archives destination. On the destination server, verify that all files have been uploaded.

Action

On the destination server, verify that all uploaded files are present.

```
% ls
R1_20111209_213452_ge-interfaces
R1_20111209_213409_juniper.conf.gz
R1_20111209_212941_messages
R1_20111209_212619_policy1-script-output
```

Meaning

Note that the filename format for each file includes the device name, the filename, and the date and time stamp.

If all of the uploaded files are present, the event policy and upload actions are working correctly. If none of the files are uploaded, verify that the destination is configured and that the archive site URL and any required password is entered correctly. For information about configuring destinations, see "Example: Define Destinations for File Archiving by Event Policies" on page 1062. If a portion of the files are missing, configure a longer transfer delay and increase the retry count and retry interval for those files.

RELATED DOCUMENTATION

Example: Define Destinations for File Archiving by Event Policies | 1062

Example: Configuring the Delay Before Files Are Uploaded by an Event Policy

When an event policy action generates output files, you can archive the files for later analysis. Similarly, you might want to archive system files, including system log files, core files, and configuration files, from the time an event occurs. You can configure an event policy to upload relevant files to a specified location for analysis. By default, the event policy immediately uploads the files. However, you can configure a transfer delay, which causes the event process (eventd) to wait a specified number of seconds before beginning to upload one or more files. The transfer delay helps ensure that a large file, such as a core file, is completely generated before the upload begins. The following event policies are configured with a transfer delay:

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 {
            "scp://robot@my.little.com/foo/moo" password "password";
            "scp://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 {
            "scp://robot@my.little.com/foo/moo" password "password";
            "scp://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 {
            "scp://robot@my.little.com/foo/moo" password "password";
            "scp://robot@my.big.com/foo/moo" password "password";
        }
    }
}
```

Example: Configuring an Event Policy to Retry the File Upload Action

When an event policy action generates output files, you can archive the files for later analysis. Similarly, you might want to archive system files, including system log files, core files, and configuration files, from the time an event occurs. You can configure an event policy to upload relevant files to a specified location for analysis. By default, if the file upload operation fails for any reason, the event policy does not retry the upload operation. However, you can configure an event policy to retry the file upload operation a specified number of times if the initial upload fails. You can also configure the time interval between each retry attempt. The following event policies are configured to retry the file upload operation:

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.

CHAPTER 38

Configure Event Policy Privileges

IN THIS CHAPTER

Change the User Privilege Level for an Event Policy Action

Only superusers can configure event policies. By default, event policy actions—such as executing operational mode commands, uploading files, and executing SLAX and XSLT event scripts—are executed by user `root`, because the event process (eventd) runs with root privileges.

NOTE: To prevent the execution of unauthorized Python code on devices running Junos OS, by default, Junos OS executes Python event scripts using the access privileges of the generic, unprivileged user and group `nobody`.

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, configure the `user-name` statement.

```
user-name username;
```

You can include this statement at the following hierarchy levels:

- `[edit event-options policy policy-name then change-configuration]`

- [edit event-options policy *policy-name* then event-script *filename*]

> **NOTE**: The user-name statement only applies to SLAX and XSLT event scripts. This statement has no effect when configured for Python event scripts.

- [edit event-options policy *policy-name* then execute-commands]

> **NOTE**: If you include the op url command to execute a remote script as an event policy action, Python scripts are always executed using the access privileges of the generic, unprivileged user and group nobody. If you do not configure the user-name statement, SLAX and XSLT scripts are executed with root privileges.

- [edit event-options policy *policy-name* then upload filename (*filename* | committed) destination *destination-name*]

By default, Junos OS executes Python event scripts with the access privileges of the generic, unprivileged user and group nobody. Starting in Junos OS Release 16.1R3, you can execute a local Python event script under the access privileges of a specific user. To specify the user, configure the python-script-user *username* statement at the [edit event-options event-script file *filename*] hierarchy level.

```
[edit event-options event-script file filename]
user@host# set python-script-user username
```

> **NOTE**: To enable a user who does not belong to the file's user or group class to execute an unsigned Python automation script, the script's file permissions must include read permission for others.

> **NOTE**: The username that you specify for the user-name and python-script-user statements must be configured at the [edit system login] hierarchy level.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
16.1R3	Starting in Junos OS Release 16.1R3, you can execute a local Python event script under the access privileges of a specific user.

RELATED DOCUMENTATION

Example: Associating an Optional User with an Event Policy Action | 1080

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;
        }
```

```
    }
}
```

RELATED DOCUMENTATION

CHAPTER 39

Event Scripts Overview

IN THIS CHAPTER

Event Scripts Overview

IN THIS SECTION

Understanding Event Scripts

Junos OS event scripts are triggered automatically by defined event policies in response to a system event and can instruct 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 enable you to:

- 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

Event scripts are based on the Junos XML management protocol and the Junos XML API, which are discussed in "Junos XML Management Protocol and Junos XML API Overview" on page 14. Event scripts can be written in Python, Extensible Stylesheet Language Transformations (*XSLT*), or Stylesheet Language Alternative syntaX (SLAX) scripting language. Event scripts use *XML Path Language* (XPath) to locate the operational objects to be inspected and automation script constructs to specify the actions to perform on the operational objects. The actions can change the output or execute additional commands based on the output.

Event scripts are invoked within an event policy. To use event scripts in an event policy, event scripts must be enabled at the `[edit event-options event-script file]` hierarchy level and must be configured under the `then` hierarchy for the event policy. When an event policy is triggered, the policy forwards the event details to the event scripts and executes the scripts in the order in which they are configured. These scripts contain instructions that execute operational mode commands, inspect the output automatically, and perform any necessary actions. For information about event policies, see "Event Policies and Event Notifications Overview" on page 961 and "Execute Event Scripts in an Event Policy" on page 1021.

You can use event scripts to generate changes to the device 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.

Benefits of Event Scripts

Event scripts, in conjunction with event policies, provide the following benefits:

- Improve network reliability and maximize network uptime by automatically diagnosing and fixing problems in the network
- Shorten troubleshooting time and speed time to resolution for network issues by automating troubleshooting tasks
- Reduce the time required for manual system monitoring and intervention
- Enable you to ensure consistent event policies across devices, simplify deployment of event policies, and reduce the size of the configuration by embedding event policies directly within the event script

RELATED DOCUMENTATION

Junos XML Management Protocol and Junos XML API Overview | 14
Understanding Python Automation Scripts for Junos Devices | 271
SLAX Overview | 83
XSLT Overview | 19
Execute Event Scripts in an Event Policy | 1021

CHAPTER 40

Create and Execute Event Scripts

IN THIS CHAPTER

Required Boilerplate for Event Scripts

SUMMARY

Define the boilerplate for event scripts.

Junos OS event scripts can be written in Extensible Stylesheet Language Transformations (*XSLT*), Stylesheet Language Alternative syntaX (SLAX), or Python. Event scripts must include the necessary boilerplate required for that script language for both basic script functionality as well as any optional functionality used within the script such as the Junos OS extension functions and named templates. This topic provides standard boilerplate that can be used in XSLT, SLAX, and Python event scripts.

SLAX and XSLT 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.

XSLT Boilerplate for Event Scripts

The XSLT event script boilerplate is as follows:

```
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 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 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 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 "Understanding Named Templates in Junos OS Automation Scripts" on page 427.

```
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 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 22.

```
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

The SLAX op script boilerplate is as follows:

```
version 1.2;
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
        */
    }
}
```

Python Boilerplate for Event Scripts

Python event scripts do not have a required boilerplate, but they must import any objects that are used in the script. Python event scripts can import the following:

- `Junos_Context` dictionary—Contains information about the script execution environment.
- `Junos_Trigger_Event` and `Junos_Received_Events` objects—Contain details about the events that triggered the corresponding event policy.
- `Junos_Remote_Execution_Details`—Generator function that is required to access remote execution details configured for an event script at the `[edit event-options event-script file` *`filename`* `remote-execution]` hierarchy level.
- `jcs` library—Enables the script to use Junos OS extension functions and Junos OS named template functionality in the script.
- `jnpr.junos` module and classes—Enables the script to use Junos PyEZ.

For example:

```
from junos import Junos_Context
from junos import Junos_Trigger_Event
from junos import Junos_Received_Events
from junos import Junos_Remote_Execution_Details
from jnpr.junos import Device
import jcs

if __name__ == '__main__':
```

Python automation scripts do not need to include an interpreter directive line (`#!/usr/bin/env python`) at the start of the script. However, the program will still execute correctly if one is present.

RELATED DOCUMENTATION

Use Event and Remote Execution Details in Event Scripts

Event policy actions can include executing one or more event scripts. When an event policy executes an event script, the event process forwards the event details to the 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.

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. Trigger event details are always forwarded to event scripts. Received event details are only present when an event policy is triggered for correlated events.

Remote execution details, which include the hostname, username, and passphrase for one or more remote hosts, enable an event script to invoke remote procedure calls on remote hosts without encoding the connection information directly in the event script. You configure remote execution details at the `[edit event-options event-script file *filename* remote-execution]` hierarchy level. When you include remote execution details in the configuration instead of in the individual event scripts, the information is captured in a single location and the passphrase is encrypted. This is not the case in an event script file.

Event details and remote execution details are forwarded to SLAX and XSLT event scripts as XML in the following format:

```
<event-script-input>
    <junos-context>
        ...
    </junos-context>
    <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>
        <message>message-string</message>
        <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 the `<junos-context>` element, see "Global Parameters and Variables in Junos OS Automation Scripts" on page 327.

Python event scripts must import the Junos_Trigger_Event and Junos_Received_Events objects to access details about the trigger event and received events. Junos_Trigger_Event and Junos_Received_Events are lxml.etree _Element objects and contain the same hierarchy and tag names as the <trigger-event> and <received-events> hierarchies in the SLAX and XSLT script input.

Python event scripts can extract the necessary event details from the objects using lxml methods such as xpath() and find(), findall(), and findtext(). For example:

```
from junos import Junos_Trigger_Event
from junos import Junos_Received_Events

id = Junos_Trigger_Event.xpath('//trigger-event/id')[0].text
name = Junos_Trigger_Event.xpath('//trigger-event/process/name')[0].text
message = Junos_Trigger_Event.xpath('//trigger-event/message')[0].text
```

Python event scripts must import Junos_Remote_Execution_Details to access the remote execution details configured at the [edit event-options event-script file *filename* remote-execution] hierarchy level. Junos_Remote_Execution_Details is a generator function that produces a sequence of remote devices, which makes it easy to iterate over multiple configured hosts. You can reference the hostname, username, and passphrase for a configured remote host by using the host, user, and passwd properties as in the following code:

```
from junos import Junos_Remote_Execution_Details

for remote in Junos_Remote_Execution_Details():
    hostname = remote.host
    username = remote.user
    passphrase = remote.passwd
```

RELATED DOCUMENTATION

How to Use RPCs and Operational Mode Commands in Event Scripts | 1091

Global Parameters and Variables in Junos OS Automation Scripts | 327

Example: Limit Event Script Output Based on a Specific Event Type | 1048

How to Use RPCs and Operational Mode Commands in Event Scripts

IN THIS SECTION

Most Junos OS operational mode commands have XML equivalents. Event scripts can execute these XML commands on a local or remote device using the *remote procedure call* (RPC) protocol. All operational mode commands that have XML equivalents are listed in the *Junos XML API Operational Developer Reference*.

SLAX and XSLT scripts execute RPCs on a local or remote device by using the `jcs:invoke()` or `jcs:execute()` extension functions, respectively. In Python scripts, RPCs are easy to execute using Junos PyEZ APIs. Each instance of the Junos PyEZ `Device` class has an `rpc` property that enables you to execute any RPC available through the Junos XML API. After establishing a session with a local or remote device, you can execute the RPC by appending the `rpc` property and RPC method name to the device instance. The return value is an XML object starting at the first element under the `<rpc-reply>` tag.

Use of RPCs and operational mode commands in event scripts is discussed in more detail in the following sections:

Executing RPCs on a Local Device

In a SLAX or XSLT event script, to execute an RPC on the local device, include the RPC in a variable declaration, and call the `jcs:invoke()` extension function with the RPC variable as an argument. The following snippet invokes an RPC on the local device:

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);
```

In a Python event script, to execute an RPC on the local device, create the `Device` instance using an empty argument list, and append the `rpc` property and the RPC method name and argument list to the device instance.

Python Syntax

```
from jnpr.junos import Device

with Device() as jdev:
    rsp = jdev.rpc.get_interface_information()
```

> **NOTE**: When you create the `Device` instance using an empty argument list to connect to the local device, Junos OS uses the access privileges of the user configured at the `[edit event-options event-script file` *`filename`* `python-script-user]` hierarchy level. If the `python-script-user` statement is omitted, Junos OS uses the access privileges of the generic, unprivileged user and group `nobody`.

Executing RPCs on a Remote Device

In a SLAX or XSLT event script, to execute an RPC on a remote device, first include the RPC in a variable declaration, and create a connection handle using the `jcs:open()` extension function with the arguments required to connect to the remote device. Then call the `jcs:execute()` extension function and include the connection handle and RPC variable as arguments. For example:

XSLT Syntax

```
<xsl:variable name="rpc">
    <get-interface-information/>
</xsl:variable>
<xsl:variable name="connection"
        select="jcs:open('198.51.100.1', 'bsmith', 'test123')"/>
<xsl:variable name="out" select="jcs:execute($connection, $rpc)"/>
<xsl:value-of select="jcs:close($connection)"/>
...
```

SLAX Syntax

```
var $rpc = <get-interface-information>;
var $connection = jcs:open("198.51.100.1", "bsmith","test123");
var $out = jcs:execute($connection, $rpc);
expr jcs:close($connection);
...
```

In a Python event script, to execute an RPC on a remote device, first create an instance of `Device` using the arguments required to connect to the remote device. Then execute the RPC by appending the `rpc` property and the RPC method name and argument list to the device instance.

Python Syntax

```
from jnpr.junos import Device

with Device(host='198.51.100.1', user='bsmith', passwd='test123') as jdev:
    rsp = jdev.rpc.get_interface_information()
```

NOTE: Junos OS connects to and executes operations on the remote device using the access privileges of the user specified in the `Device()` argument list, even if a different user is configured for the `python-script-user` statement at the `[edit event-options event-script file` *`filename`*`]` hierarchy level.

To avoid adding the remote connection details directly into an event script, you can specify remote execution details for each event script that executes RPCs on a remote device at the `[edit event-options event-script file` *`filename`* `remote-execution]` hierarchy level. We recommend adding the remote execution details to the configuration instead of directly in the event script, because all of the information is available in a single location, and the passphrase is encrypted in the configuration.

For each remote device where an RPC is executed, configure the device hostname and the corresponding username and passphrase.

```
[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 executed by an event policy. For more information about the details that are forwarded to the event script, see "Use Event and Remote Execution Details in Event Scripts" on page 1088.

An event script references the remote execution details in the argument list of the function used to create the connection to the remote host. Once the connection has been established, the script can execute RPCs on that device.

In Python event scripts, you reference the remote execution details in the argument list of the Junos PyEZ `Device()` instance. The following code iterates over the remote execution details for all hosts configured for that event script and connects to and executes the same RPC on each host.

Python Syntax

```
from junos import Junos_Remote_Execution_Details
from jnpr.junos import Device

def main()
    for remote in Junos_Remote_Execution_Details():
        hostname = remote.host
        username = remote.user
        passphrase = remote.passwd

        with Device(host=hostname, user=username, passwd=passphrase) as jdev:
            inv = jdev.rpc.get_interface_information()
            #process RPC information...

if __name__ == "__main__":
    main()
```

In SLAX or XSLT scripts, create a connection to the remote host by using the `jcs:open()` function and reference the remote execution details in the argument list. For example:

XSLT Syntax

```
<xsl:variable name="rpc">
    <get-interface-information/>
</xsl:variable>
<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,$d/passphrase)"/>
    <xsl:variable name="out" select="jcs:execute($connection, $rpc)"/>
    <xsl:value-of select="jcs:close($connection)"/>
    ...
</xsl:for-each>
```

SLAX Syntax

```
var $rpc = <get-interface-information>;
for-each (event-script-input/remote-execution-details) {
    var $d = remote-execution-detail;
    var $connection = jcs:open($d/remote-hostname,$d/username,$d/passphrase);
    var $out = jcs:execute($connection, $rpc);
    expr jcs:close($connection);
     ...
}
```

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 using 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 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 (198.51.100.1)' 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 'router2,198.51.100.1' (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 (198.51.100.1)' 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 'router2,198.51.100.1' (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.

Displaying the RPC Tags for a Command

You can display the RPC XML tags for operational mode commands in the CLI of the device. To display the RPC XML tags for a 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>
```

SLAX and XSLT scripts can execute RPCs using the RPC XML tags. Python scripts must convert the RPC tags and command options into a format suitable for Python. For more information about using Junos PyEZ to execute RPCs and about mapping RPC tags to the corresponding Python method and method arguments, see Using Junos PyEZ to Execute RPCs on Devices Running Junos OS.

Using Operational Mode Commands in Event Scripts

Some operational mode commands do not have XML equivalents. SLAX and XSLT scripts can execute commands that have no XML equivalent by using the `<command>` element. Python scripts can execute these commands by using the Junos PyEZ `cli()` method defined in the `Device` class.

If a command is not listed in the *Junos XML API Operational Developer 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 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 similar to 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 Developer Reference* is usually more reliable.

SLAX and XSLT event scripts can execute 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: Display DNS Hostname Information Using an Op Script" on page 892.

```
<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>
...
```

Python event scripts can execute commands that have no XML equivalent by using Junos PyEZ APIs. The `cli()` method defined in the `Device` class executes an operational mode command and returns the output in text format. For example:

```
from jnpr.junos import Device

def main():
    with Device() as jdev:
        res = jdev.cli('show host hostname',  warning=False)
        print (res)
```

```
if __name__ == "__main__":
    main()
```

You can also specify `format='xml'` to return the output formatted as Junos OS XML elements. For more information about the Junos PyEZ `cli` method, see http://junos-pyez.readthedocs.org/en/latest/_modules/jnpr/junos/device.html#Device.cli .

RELATED DOCUMENTATION

Enable and Execute Event Scripts

Event scripts are stored on a device's hard disk 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 "Store and Enable Junos Automation Scripts" on page 448 and "Store Scripts in Flash Memory" on page 451.

To prevent the execution of unauthorized Python code on devices running Junos OS, unsigned Python scripts must meet certain requirements before you can execute the scripts on a device. Starting in Junos OS Release 16.1R3, unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file. Prior to Junos OS Release 16.1R3, unsigned Python scripts must only be owned by the root user. For detailed information about the requirements for executing Python automation scripts on devices running Junos OS, see "Requirements for Executing Python Automation Scripts on Junos Devices" on page 276.

NOTE: If the device has dual Routing Engines and you want to enable an event script to execute on both Routing Engines, you can copy the script to the **/var/db/scripts/event** or **/config/scripts/event** directory on both Routing Engines, or you can issue the `commit synchronize scripts` command to synchronize the configuration and copy the scripts to the other Routing Engine as part of the commit operation.

You must enable an event script before it can be executed. To enable an event script, include the `file` *`filename`* statement at the `[edit event-options events-script]` hierarchy level, and specify the name of the file containing the event script. Only users who belong to the Junos `super-user` login class can enable event scripts.

```
[edit event-options event-script]
user@host# set file filename
```

SLAX and Python scripts must include the **.slax** or **.py** filename extension, respectively, in both the actual script name and the filename in the configuration. XSLT scripts do not require a filename extension, 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 event-options event-script file]` 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.

By default, you cannot execute unsigned Python scripts on devices running Junos OS. To enable the execution of unsigned Python automation scripts that meet the requirements outlined in "Requirements for Executing Python Automation Scripts on Junos Devices" on page 276, you must configure the `language python` or `language python3` statement at the `[edit system scripts]` hierarchy level.

```
[edit system scripts]
user@host# set language (python | python3)
```

By default, Junos OS executes Python event scripts with the access privileges of the generic, unprivileged user and group `nobody`. Starting in Junos OS Release 16.1R3, you can specify the user under whose access privileges the Python script will execute. To execute a Python event script under the access privileges of a specific user, configure the `python-script-user` *`username`* statement at the `[edit event-options event-script file` *`filename`*`]` hierarchy level.

```
[edit event-options event-script file filename]
user@host# set python-script-user username
```

NOTE: To enable a user who does not belong to the file's user or group class to execute an unsigned Python automation script, the script's file permissions must include read permission for others.

To determine which event scripts are currently enabled on the device, use the `show` command to display the files configured at the `[edit event-options event-script]` hierarchy level.

```
[edit event-options event-script]
user@host# show
```

To ensure that the enabled files are on the device, list the contents of the **/var/run/scripts/event/** directory using the `file list /var/run/scripts/event` operational mode command.

```
user@host> file list /var/run/scripts/event
```

When you issue the `commit` command, event scripts configured 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 policy can execute an event script in response to an event notification.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
16.1R3	Starting in Junos OS Release 16.1R3, unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

RELATED DOCUMENTATION

Event Scripts Overview | 1082

Execute Event Scripts in an Event Policy | 1021

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/event** directory on the device.

3. Run the script through one or more hash functions to calculate hash values.

   Starting in Junos OS Release 18.2R2 and 18.3R1, Junos OS supports only the SHA-256 hash function for configuring script checksum hashes. Earlier releases support the *MD5*, *SHA-1*, and SHA-256 hash functions.

```
user@host> file checksum md5 /var/db/scripts/event/script1.slax
MD5 (/var/db/scripts/event/script1.slax) = 3af7884eb56e2d4489c2e49b26a39a97
```

```
user@host> file checksum sha1 /var/db/scripts/event/script1.slax
SHA1 (/var/db/scripts/event/script1.slax) = 00dc690fb08fb049577d012486c9a6dad34212c0
```

```
user@host> file checksum sha-256 /var/db/scripts/event/script1.slax
SHA256 (/var/db/scripts/event/script1.slax) =
150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
```

4. Configure the script and the `checksum` statement for one or more hash values.

```
[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, Junos OS recalculates the checksum value using the configured hash algorithm 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.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
18.3R1	Starting in Junos OS Release 18.2R2 and 18.3R1, Junos OS supports only the SHA-256 hash function for configuring script checksum hashes.

RELATED DOCUMENTATION

Configure Checksum Hashes for a Commit Script | **564**

Configure Checksum Hashes for an Op Script | **849**

Configuring Checksum Hashes for an SNMP Script | **1130**

checksum

Replace 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 the existing event script or write a new one.
2. Copy the script to the **/var/db/scripts/event** directory on the hard disk or the **/config/scripts/event** directory on the flash drive; for information about setting the storage location for scripts, see "Store 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**: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

> **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. Alternatively, you can synchronize scripts between Routing Engines by using the `commit synchronize scripts` command when you commit the configuration.

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.

RELATED DOCUMENTATION

Store Scripts in Flash Memory | 451

CHAPTER 41

Troubleshoot Event Policies and Event Scripts

IN THIS CHAPTER

Trace Event Policy Processing on Devices Running Junos OS

IN THIS SECTION

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:

- Events are logged to the **/var/log/eventd** file on the device.
- 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 System Log Explorer.)

- 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

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, no 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 61 on page 1108 describes the meaning of the event policy tracing flags.

Table 61: 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

Table 61: Event Policy Tracing Flags *(Continued)*

Flag	Description	Default Setting
`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.

```
user@host> show log eventd | last
```

RELATED DOCUMENTATION

Understanding the Event System Log Priority in an Event Policy | 1040
Trace Event Script Processing on Devices Running Junos OS | 1112
traceoptions

Trace Event Policy Processing on Devices Running Junos OS Evolved

IN THIS SECTION

An event policy is automatically triggered upon receipt of certain events, and thus it does not direct output to the terminal. To get information about the execution of event policies, you can review the event policy trace data.

Junos OS Evolved captures trace data for all applications on all nodes on the Routing Engine by default. Event policy tracing tracks and logs all event policy operations. You can view the collected traces for information about the processing of event policies, including the executed policies, their trigger events, the corresponding event policy actions, and any additional operations, warnings, and errors.

How to Display Trace Data for Event Policies

Junos OS Evolved stores the trace data from all nodes that is collected on the primary Routing Engine under the **/var/log/traces** directory. The `master-eventd` application handles event policy, and the event policy trace data is stored under the ***node*.master-eventd.*sequence-number*** subdirectories.

To view the event policy trace data, issue the `show trace application master-eventd` operational mode command.

```
user@host> show trace application master-eventd
2021-06-16 13:09:47.320304999 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_33  msg = "policy should now be executed",
policy = "backup_config"
2021-06-16 13:09:47.320324689 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_21  msg = "Executing Policy", policy =
"backup_config"
2021-06-16 13:09:47.320348369 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_1  func_name = "eventd_policy_action_next"
2021-06-16 13:09:47.320365189 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_1  func_name = "eventd_policy_action_next"
2021-06-16 13:09:47.320372819 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_1  func_name = "eventd_policy_action_next"
2021-06-16 13:09:47.320438179 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_6  msg = "executing commands", PolicyName
= "backup_config", evt_id = "UI_COMMIT_COMPLETED", cmd_file = "/tmp/evt_cmd_vfEgUQ", output_file
= "/tmp/evt_op_gDnEpr"
2021-06-16 13:09:47.320464959 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_1  func_name =
"eventd_substitute_variable_cmd"
2021-06-16 13:09:49.150642247 re0:master-eventd:10911:TRACE_INFO
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_38  msg = "Eventd Rcvd signal", signo = 17
2021-06-16 13:09:49.150656107 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_27  msg = "SIGCHLD received"
2021-06-16 13:09:49.150687637 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_28  msg = "SIGCHLD received", pid = 492
2021-06-16 13:09:49.150696647 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_29  msg = "Child terminated", pid = 492,
```

```
status = "successfully"
2021-06-16 13:09:49.150716687 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_1  func_name = "eventd_policy_action_next"
2021-06-16 13:09:49.150735047 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_1  func_name = "eventd_upload_file"
2021-06-16 13:09:49.158191526 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_28  msg = "SIGCHLD received", pid = 0
2021-06-16 13:09:49.354883114 re0:master-eventd:10911:TRACE_INFO
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_38  msg = "Eventd Rcvd signal", signo = 17
2021-06-16 13:09:49.354901704 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_27  msg = "SIGCHLD received"
2021-06-16 13:09:49.354924204 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_28  msg = "SIGCHLD received", pid = 506
2021-06-16 13:09:49.354932034 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_29  msg = "Child terminated", pid = 506,
status = "successfully"
2021-06-16 13:09:49.354944754 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_1  func_name = "eventd_policy_action_next"
2021-06-16 13:09:49.354989194 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_1  func_name = "eventd_policy_action_next"
2021-06-16 13:09:49.354998454 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_1  func_name = "eventd_policy_action_next"
2021-06-16 13:09:49.355005984 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_3  msg = "removing active policy",
PolicyName = "backup_config"
2021-06-16 13:09:49.355115774 re0:master-eventd:10911:TRACE_ERR
EVENTOPTIONS_TRACE_POLICY_P_EVENTOPTIONS_TRACE_POLICY_28  msg = "SIGCHLD received", pid = -1
...
```

You can include the `terse` option to display just the timestamp and message.

```
user@host> show trace application master-eventd terse
2021-06-16 13:09:47.320304999 msg = "policy should now be executed", policy = "backup_config"
2021-06-16 13:09:47.320324689 msg = "Executing Policy", policy = "backup_config"
2021-06-16 13:09:47.320348369 func_name = "eventd_policy_action_next"
2021-06-16 13:09:47.320365189 func_name = "eventd_policy_action_next"
2021-06-16 13:09:47.320372819 func_name = "eventd_policy_action_next"
2021-06-16 13:09:47.320438179 msg = "executing commands", PolicyName = "backup_config", evt_id =
"UI_COMMIT_COMPLETED", cmd_file = "/tmp/evt_cmd_vfEgUQ", output_file = "/tmp/evt_op_gDnEpr"
2021-06-16 13:09:47.320464959 func_name = "eventd_substitute_variable_cmd"
...
```

How to Modify Trace Settings for Event Policies

Junos OS Evolved traces event policy processing by default and traces all applications at the info level for informational messages. You can configure trace settings for specific applications at the `[edit system application]` hierarchy level. For example, you can specify the trace level of the application on a given node.

To modify event policy tracing operations, configure the settings under the `[edit system trace application master-eventd]` hierarchy level. The following example configures tracing for the `master-eventd` application at the notice level on node re0:

```
[edit]
user@host# set system trace application master-eventd node re0 level notice
user@host# commit
```

For more information about configuring trace settings, see *trace*.

Trace Event Script Processing on Devices Running Junos OS

IN THIS SECTION

Event script tracing operations track 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, which include errors, warnings, progress messages, and script processing events, in the **/var/log/escript.log** file on the device. 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.

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. Commit the configuration.

```
[edit]
user@host# commit and-quit
```

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" on page 1113 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:

```
user@host> show log escript.log | last
```

Table 62 on page 1114 summarizes useful filtering commands that display selected portions of the **escript.log** file.

Table 62: 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 and-quit
```

```
user@host> 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, the traceoptions `events` flag is turned on, regardless of the configuration settings, and only important events are logged. This includes errors, warnings, progress messages, and script processing events. 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 63 on page 1117 describes the meaning of the event script tracing flags.

Table 63: Event Script Tracing Flags

Flag	Description	Default Setting
`all`	Trace all operations.	Off
`events`	Trace important events, including errors, warnings, progress messages, and script processing 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
`xslt`	Trace the Extensible Stylesheet Language Transformations (XSLT) library.	Off

RELATED DOCUMENTATION

Enable and Execute Event Scripts | 1099

Trace Event Policy Processing on Devices Running Junos OS | 1105

traceoptions

Trace Script Processing on Devices Running Junos OS Evolved

IN THIS SECTION

When you execute an interactive script, the script can generate output, including warnings and errors, in the CLI or RPC reply. When the system triggers non-interactive scripts, for example, when an event policy triggers an event script, the script does not direct output to the terminal. In either case, you might need more information about the execution of the script. Junos OS Evolved captures trace data for all applications by default. You can view the collected traces for additional script processing information, including the memory and CPU usage, script arguments, script execution, and warnings and errors.

Junos OS Evolved collects trace data from all applications on all nodes on the Routing Engine. Whereas Junos OS logs the trace data for each type of script in separate log files, Junos OS Evolved stores the trace data for all scripts in the same location. The trace log includes data for commit, event, op, and SNMP scripts; YANG action and translation scripts; and Juniper Extension Toolkit scripts.

How to Display Trace Data for Scripts

Junos OS Evolved stores the trace data from all nodes that is collected on the primary Routing Engine under the **/var/log/traces** directory. The `cscript` application handles scripts, and the trace data for scripts is stored under the ***node*.cscript.*sequence-number*** subdirectories.

To view trace data for scripts, issue the `show trace application cscript` operational mode command.

```
user@host> show trace application cscript
2021-05-20 09:11:42.239695672 re0:cscript:4176:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg =
"Process's current softlimit [134217728] hardlimit [-1]"
2021-05-20 09:11:42.239773157 re0:cscript:4176:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg =
"Process's limits are already set by parent process"
2021-05-20 09:11:42.239812430 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"op script processing begins"
2021-05-20 09:11:42.239855140 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: /usr/libexec/ui/cscript"
2021-05-20 09:11:42.239865140 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -mop"
2021-05-20 09:11:42.239866196 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -p"
2021-05-20 09:11:42.239867156 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: /"
2021-05-20 09:11:42.239868116 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -Q2"
```

```
2021-05-20 09:11:42.239869131 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -f"
2021-05-20 09:11:42.239882048 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: hello.py"
2021-05-20 09:11:42.239883202 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -d134217728"
2021-05-20 09:11:42.239884135 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -E"
2021-05-20 09:11:42.239885131 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg:  user admin logname admin host host tty /dev/pts/0 agent op-script current-
directory /var/home/admin pid 32212 ppid 32206"
2021-05-20 09:11:42.239886175 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -u"
2021-05-20 09:11:42.239887176 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: admin"
2021-05-20 09:11:42.239888251 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -U"
2021-05-20 09:11:42.239889287 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -i9"
2021-05-20 09:11:42.245988806 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"running op script 'hello.py'"
2021-05-20 09:11:42.246006519 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"opening op script '/var/db/scripts/op/hello.py'"
...
```

You can include the `terse` option to display just the timestamp and message.

```
user@host> show trace application cscript terse
2021-05-20 09:11:42.239695672 msg = "Process's current softlimit [134217728] hardlimit [-1]"
2021-05-20 09:11:42.239773157 msg = "Process's limits are already set by parent process"
2021-05-20 09:11:42.239812430 msg = "op script processing begins"
...
```

You can also refine the traces to display by specifying the trace time elapsed, process ID, and node. For example, the following command shows trace data for a specific process ID.

```
user@host> show trace application cscript pid 10683
2021-05-24 09:42:09.552687492 re0:cscript:10683:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg
= "Process's current softlimit [134217728] hardlimit [-1]"
2021-05-24 09:42:09.552819712 re0:cscript:10683:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg
= "Process's limits are already set by parent process"
```

```
2021-05-24 09:42:09.552897412 re0:cscript:10683:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg
= "action script processing begins"
2021-05-24 09:42:09.553025992 re0:cscript:10683:TRACE_INFO
CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1  msg = "arg: /usr/libexec/ui/cscript"
2021-05-24 09:42:09.553095062 re0:cscript:10683:TRACE_INFO
CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1  msg = "arg: -maction"
...
```

How to Modify Trace Settings for Scripts

Junos OS Evolved traces script processing by default and traces all applications at the info level for informational messages. You can configure trace settings for specific applications at the `[edit system application]` hierarchy level. For example, you can specify the trace level of the application on a given node.

To modify script tracing operations, configure the settings under the `[edit system trace application cscript]` hierarchy level. The following example configures the `cscript` application to trace script processing on node re0 at the debug level:

```
[edit]
user@host# set system trace application cscript node re0 level debug
user@host# commit
```

For more information about configuring trace settings, see *trace*.

11

PART

SNMP Scripts

CHAPTER 42

SNMP Scripts Overview

IN THIS CHAPTER

SNMP Scripts Overview

Junos OS SNMP scripts, which are supported in Junos OS Release 15.1 and later releases, provide the flexibility to support custom MIBs that are not implemented in the Junos operating system (Junos OS). SNMP scripts are triggered automatically when the SNMP manager requests information from the SNMP agent for an object identifier (OID) that is mapped to an SNMP script for an unsupported OID. The script acts like an SNMP subagent, and the system sends the return value from the script to the network management system (NMS).

NOTE: SNMP must be configured on the device before SNMP scripts can provide values for OIDs.

Table 64 on page 1122 lists SNMP script tasks and where to find details.

Table 64: SNMP Script Tasks

SNMP Script Task	Find Details Here
Getting started with the SLAX script language	• "SLAX Overview" on page 83 • "Understanding Extension Functions in Junos OS Automation Scripts" on page 337
Developing SNMP scripts	• "Required Boilerplate for SNMP Scripts" on page 1124 • "Example: Processing Unsupported OIDs with an SNMP Script" on page 1133

Table 64: SNMP Script Tasks *(Continued)*

SNMP Script Task	Find Details Here
Configuring CLI for SNMP scripts	• "Enabling an SNMP Script" on page 1128 • *snmp*
Debugging SNMP scripts	• "Tracing SNMP Script Processing" on page 1140
Managing SNMP scripts	• "Configuring and Using a Master Source Location for a Script" on page 457 • "Using an Alternate Source Location for a Script" on page 467 • "Synchronizing Scripts Between Routing Engines" on page 514 • "Storing Scripts in Flash Memory" on page 451 • "Understanding Limits on Executed Event Policies and Memory Allocation for Scripts" on page 502
Adding security for SNMP scripts	• "Configuring Checksum Hashes for an SNMP Script" on page 1130

SNMP scripts are based on the Junos XML management protocol and the Junos XML API, which are discussed in "Junos XML Management Protocol and Junos XML API Overview" on page 14. You can write SNMP scripts in Python, Extensible Stylesheet Language Transformations (*XSLT*), or Stylesheet Language Alternative syntaX (SLAX). SNMP scripts written in SLAX and XSLT 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.

RELATED DOCUMENTATION

CHAPTER 43

Create and Execute SNMP Scripts

IN THIS CHAPTER

Required Boilerplate for SNMP Scripts

Junos OS SNMP scripts can be written in Extensible Stylesheet Language Transformations (*XSLT*), Stylesheet Language Alternative syntaX (SLAX), or Python. SNMP scripts must include the necessary boilerplate required for that script language for both basic script functionality, as well as any optional functionality used within the script such as the Junos OS extension functions and named templates. This topic provides standard boilerplate that can be used in XSLT, SLAX, and Python SNMP scripts.

SLAX Boilerplate for SNMP Scripts

The SLAX SNMP script boilerplate is as follows:

```
version 1.2;

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 dyn = "http://exslt.org/dynamic";
ns snmp extension = "http://exslt.org/functions";

match / {
    <snmp-script-results> {
        var $snmp-action = snmp-script-input/snmp-action;
        var $snmp-oid = snmp-script-input/snmp-oid;
           <!-- ... insert your code here ... -->
        <snmp-oid> $snmp-oid;
```

```
12            <snmp-type> $snmp-type;
13            <snmp-value> $snmp-value;
            }
        }
```

NOTE: SLAX and XSLT SNMP 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.

Line 1 specifies the SLAX version as 1.2.

```
1    version 1.2;
```

Lines 2 through 6 list all the namespace mappings commonly used in SNMP scripts. Not all of these prefixes are used in this example. Listing all namespace mappings prevents errors if the mappings are used in later versions of the script. These namespace mappings enable you to use extension functions and named templates in your scripts. These extension functions and named templates are discussed in "Understanding Extension Functions in Junos OS Automation Scripts" on page 337 and "Understanding Named Templates in Junos OS Automation Scripts" on page 427.

Line 5 and line 6 have EXSLT namespace mappings. SNMP extension functions are defined in the namespace with the associated URI http://exslt.org/functions. Line 6 registers the `snmp` extension namespace with the EXSLT functions namespace, allowing you to define customized functions using `snmp` as a prefix within your SLAX script. For more information about the EXSLT namespace, see http://exslt.org/func/index.html.

```
2    ns junos = "http://xml.juniper.net/junos/*/junos";
3    ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
4    ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
5    ns dyn = "http://exslt.org/dynamic";
6    ns snmp extension = "http://exslt.org/functions";
```

Line 7 defines an unnamed template, `match /`, that represents the top level of the configuration hierarchy. All *XML Path Language* (XPath) expressions in the script must start at the top-level element in the configuration hierarchy. 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 22.

```
7    match / {
```

After the match / tag element, the <snmp-script-results> container tag must be the top-level child tag, as shown in Line 8. The value of this container is returned to the OID requester.

```
8        <snmp-script-results> {
```

Lines 9 and 10 define variables based on the corresponding elements that you can use in your code to determine whether the action is get or get-next, and the value of the OID.

```
9            var $snmp-action = snmp-script-input/snmp-action;
10           var $snmp-oid = snmp-script-input/snmp-oid;
```

Between Line 10 and Line 11, you can define additional code that includes XSLT templates that are called from within the match / template.

Lines 11 through 13 define the values returned by the SNMP script to the OID requester. The value of <snmp-oid> is taken from the input value of snmp-script-input/snmp-oid. For the SNMP script feature, the following object identifier types for <snmp-type> are supported:

- Counter32
- Counter64
- Integer32
- Unsigned32
- Octet String

You set the <snmp-value> to the return value from the script.

```
11           <snmp-oid> $snmp-oid;
12           <snmp-type> $snmp-type;
13           <snmp-value> $snmp-value;
```

XSLT Boilerplate for SNMP Scripts

The XSLT boilerplate is:

```
<?xml version="1.0" encoding="UTF-8" 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:dyn="http://exslt.org/dynamic"
    xmlns:snmp="http://exslt.org/functions"
    version="1.0" extension-element-prefixes="snmp">

    <xsl:template match="/">
        <snmp-script-results>
            <xsl:variable name="snmp-action" select="snmp-script-input/snmp-action"/>
            <xsl:variable name="snmp-oid" select="snmp-script-input/snmp-oid"/>
            <!-- Insert your code here -->
            <snmp-oid>
                <xsl:value-of select="$snmp-oid"/>
            </snmp-oid>
            <snmp-type>
                <xsl:value-of select="$snmp-type"/>
            </snmp-type>
            <snmp-value>
                <xsl:value-of select="$snmp-value"/>
            </snmp-value>
        </snmp-script-results>
    </xsl:template>
</xsl:stylesheet>
```

Python Boilerplate for SNMP Scripts

Python SNMP scripts do not have a required boilerplate, but they must import any objects that are used in the script. Python SNMP scripts must include the `import jcs` statement in order to use the `get_snmp_action()`, `get_snmp_oid()`, and `emit_snmp_attributes()` functions that retrieve the action and OID values passed to the script and return the data for the requested MIB object.

```
import jcs

if __name__ == '__main__':
```

Python automation scripts do not need to include an interpreter directive line (`#!/usr/bin/env python`) at the start of the script. However, the program will still execute correctly if one is present.

RELATED DOCUMENTATION

Example: Process Unsupported OIDs with an SNMP Script | **1133**

Understanding Extension Functions in Junos OS Automation Scripts | **337**

Enable an SNMP Script

SNMP scripts are stored on a device's hard drive in the **/var/db/scripts/snmp** directory or on the flash drive in the **/config/scripts/snmp** 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 and Enabling Scripts" on page 448 and "Storing Scripts in Flash Memory" on page 451.

To prevent the execution of unauthorized Python code on devices running Junos OS, unsigned Python scripts must meet certain requirements before you can execute the scripts on a device. Starting in Junos OS Release 16.1R3, unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file. Prior to Junos OS Release 16.1R3, unsigned Python scripts must only be owned by the root user. For detailed information about the requirements for executing Python automation scripts on devices running Junos OS, see "Requirements for Executing Python Automation Scripts on Devices Running Junos OS" on page 276.

NOTE: If the device has dual Routing Engines and you want to enable an SNMP script to execute on both Routing Engines, you can copy the script to the **/var/db/scripts/snmp** or **/config/scripts/snmp** directory on both Routing Engines, or you can issue the `commit synchronize scripts` command to synchronize the configuration and copy the scripts to the other Routing Engine as part of the commit operation.

You must enable an SNMP script before it can be executed. To enable an SNMP script, include the `file` *`filename`* statement at the `[edit system scripts snmp]` hierarchy level, and specify the name of a file containing an SNMP script. Only users who belong to the Junos OS `super-user` login class can enable SNMP scripts.

```
[edit system scripts snmp]
user@host# set file filename
```

SLAX and Python scripts must include the **.slax** or **.py** filename extension, respectively, in both the actual script name and the filename in the configuration. XSLT scripts do not require a filename extension, 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 file]` 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.

By default, you cannot execute unsigned Python scripts on devices running Junos OS. To enable the execution of unsigned Python automation scripts that meet the requirements outlined in "Requirements for Executing Python Automation Scripts on Devices Running Junos OS" on page 276, you must configure the `language python` or `language python3` statement at the `[edit system scripts]` hierarchy level.

```
[edit system scripts]
user@host# set language (python | python3)
```

By default, Junos OS executes Python SNMP scripts with the access privileges of the generic, unprivileged user and group `nobody`. Starting in Junos OS Release 16.1R3, you can specify the user under whose access privileges the Python script will execute. To execute a Python SNMP script under the access privileges of a specific user, configure the `python-script-user` *`username`* statement at the `[edit system scripts snmp file` *`filename`*`] hierarchy level`.

```
[edit system scripts snmp file filename]
user@host# set python-script-user username
```

NOTE: To enable a user who does not belong to the file's user or group class to execute an unsigned Python automation script, the script's file permissions must include read permission for others.

To determine which SNMP scripts are currently enabled on the device, use the `show` command to display the files configured at the `[edit system scripts snmp]` hierarchy level.

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

To ensure that the enabled files are on the device, list the contents of the **/var/run/scripts/snmp/** directory using the `file list /var/run/scripts/snmp` operational mode command.

```
user@host> file list /var/run/scripts/snmp
```

In order for SNMP scripts to return values for OIDs, SNMP must be configured on the device running Junos OS. For more information about configuring SNMP, see the Junos OS Network Management Administration Guide for Routing Devices.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
16.1R3	Starting in Junos OS Release 16.1R3, unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

Configuring Checksum Hashes for an SNMP Script

You can configure one or more checksum hashes that can be used to verify the integrity of an SNMP 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/snmp** directory on the device.
3. Run the script through one or more hash functions to calculate hash values.

Starting in Junos OS Release 18.2R2 and 18.3R1, Junos OS supports only the SHA-256 hash function for configuring script checksum hashes. Earlier releases support the *MD5*, *SHA-1*, and SHA-256 hash functions.

```
user@host> file checksum md5 /var/db/scripts/snmp/script1.slax
MD5 (/var/db/scripts/snmp/script1.slax) = 3af7884eb56e2d4489c2e49b26a39a97
```

```
user@host> file checksum sha1 /var/db/scripts/snmp/script1.slax
SHA1 (/var/db/scripts/snmp/script1.slax) = 00dc690fb08fb049577d012486c9a6dad34212c0
```

```
user@host> file checksum sha-256 /var/db/scripts/snmp/script1.slax
SHA256 (/var/db/scripts/snmp/script1.slax) =
150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
```

4. Configure the script and the `checksum` statement for one or more hash values.

```
[edit system scripts snmp]
user@host# set file script1.slax checksum md5 3af7884eb56e2d4489c2e49b26a39a97
```

```
[edit system scripts snmp]
user@host# set file script1.slax checksum sha-1 00dc690fb08fb049577d012486c9a6dad34212c0
```

```
[edit system scripts snmp]
user@host# set file script1.slax checksum
sha-256 150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
```

During the execution of the script, Junos OS recalculates the checksum value using the configured hash algorithm 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.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
18.3R1	Starting in Junos OS Release 18.2R2 and 18.3R1, Junos OS supports only the SHA-256 hash function for configuring script checksum hashes.

RELATED DOCUMENTATION

Configuring Checksum Hashes for an Event Script | 1101
Configure Checksum Hashes for an Op Script | 849
Configure Checksum Hashes for a Commit Script | 564

CHAPTER 44

SNMP Script Example

IN THIS CHAPTER

Example: Process Unsupported OIDs with an SNMP Script

IN THIS SECTION

This sample SNMP script shows how to process object identifiers (OIDs) that are not supported on devices running Junos OS.

Requirements

Junos OS Release 15.1 or later when using SLAX SNMP scripts.

Junos OS Release 16.1R3 or later when using Python SNMP scripts on QFX Series switches or MX Series, PTX Series, or T Series routers.

Junos OS Release 17.1R1 or later when using Python SNMP scripts on EX Series switches.

Junos OS Release 17.3R1 or later when using Python SNMP scripts on SRX Series Firewalls.

SNMP is configured on the device.

Overview and SNMP Script

In this example, two equivalent SNMP scripts are presented in SLAX and Python that match and process several unsupported OIDs. The script returns the value for the requested object by using the `<snmp-script-results>` element in the SLAX script and the `jcs.emit_snmp_attributes()` function in the equivalent Python script. The `syslog()` extension function is called to log the requested SNMP action and OID in the system log file. For more information about the `syslog()` function, see "syslog() Function (Python, SLAX, and XSLT)" on page 420.

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";
ns dyn = "http://exslt.org/dynamic";
ns snmp extension = "http://exslt.org/functions";

match / {

    var $snmp-action = snmp-script-input/snmp-action;
    var $snmp-oid = snmp-script-input/snmp-oid;

    expr jcs:syslog(8, "snmp-action = ", $snmp-action, " snmp-oid = ", $snmp-oid);

    if ($snmp-action == 'get') {

        if($snmp-oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.1') {
            <snmp-script-results> {
                <snmp-oid> $snmp-oid;
                <snmp-type> "Integer32";
                <snmp-value> "211";
            }
        }
        else if($snmp-oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.2') {
            <snmp-script-results> {
                <snmp-oid> $snmp-oid;
                <snmp-type> "Integer32";
                <snmp-value> "429";
            }
        }
    }
```

```
    else if ($snmp-action == 'get-next') {

        if($snmp-oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1') {
            <snmp-script-results> {
                <snmp-oid> ".1.3.6.1.4.1.2636.13.61.1.9.1.1.1";
                <snmp-type> "Integer32";
                <snmp-value> "211";
            }
        }
        else if ($snmp-oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.1') {
            <snmp-script-results> {
                <snmp-oid> ".1.3.6.1.4.1.2636.13.61.1.9.1.1.2";
                <snmp-type> "Integer32";
                <snmp-value> "429";
            }
        }
    }
}
```

Python Syntax

```
import jcs

def main():

    snmp_action = jcs.get_snmp_action()
    snmp_oid = jcs.get_snmp_oid()

    jcs.syslog("8", "snmp_action = ", snmp_action, " snmp_oid = ", snmp_oid)

    if snmp_action == 'get':
        if snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.1':
            jcs.emit_snmp_attributes(snmp_oid, "Integer32", "211")
        elif snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.2':
            jcs.emit_snmp_attributes(snmp_oid, "Integer32", "429")

    elif snmp_action == 'get-next':
        if snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1':
            jcs.emit_snmp_attributes(".1.3.6.1.4.1.2636.13.61.1.9.1.1.1", "Integer32", "211")
        elif snmp_oid == '.1.3.6.1.4.1.2636.13.61.1.9.1.1.1':
            jcs.emit_snmp_attributes(".1.3.6.1.4.1.2636.13.61.1.9.1.1.2", "Integer32", "429")
```

```
if __name__ == '__main__':
    main()
```

Configuration

IN THIS SECTION

Procedure

Step-by-Step Procedure

To download and enable the script:

> **NOTE**: You can create SNMP scripts in Python, SLAX, or XSLT. You can use the *request system scripts convert* command to convert between SLAX and XSLT.

1. Copy the script into a text file, name the file **sample_snmp.slax** or **sample_snmp.py**, as appropriate, and download it to the **/var/db/scripts/snmp** directory on the device.

   > **NOTE**: Unsigned Python scripts must be owned by either root or a user in the Junos OS `super-user` login class, and only the file owner can have write permission for the file.

2. Enable the SNMP script and configure the OID.

   In configuration mode, configure the `file` *`filename`* statement with the appropriate filename and extension for your script language at the `[edit system scripts snmp]` hierarchy level, and configure the OID that will trigger the script.

   ```
   [edit system scripts]
   user@host# set snmp file sample_snmp.slax oid .1.3.6.1.4.1.2636.13.61.1.9.1.1
   ```

3. If the script is written in Python, configure the `language python` or `language python3` statement as appropriate.

```
[edit system scripts]
user@host# set language python3
```

4. If the script is written in Python, configure the user under whose access privileges the script executes.

```
[edit system scripts]
user@host# set snmp file sample_snmp.py python-script-user username
```

NOTE: If you do not configure the `python-script-user` statement, then by default, Junos OS executes Python SNMP scripts under the access privileges of the user and group `nobody`.

5. Issue the `commit` command to commit the configuration.

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

Results

From configuration mode, confirm your configuration by entering the `show system scripts snmp` command.

```
[edit]
user@host# show system scripts snmp
file sample_snmp.slax {
    oid .1.3.6.1.4.1.2636.13.61.1.9.1.1;
}
```

If the output does not display the intended configuration, repeat the configuration instructions in this example to correct it.

To ensure that the enabled files are on the device, list the contents of the **/var/run/scripts/snmp** directory using the `file list /var/run/scripts/snmp` operational mode command.

Verification

IN THIS SECTION

Verifying the Script Execution

Purpose

Verify that the SNMP script functions as expected.

Action

Issue the `show snmp mib get`, `show snmp mib get-next`, or `show snmp mib walk` command to generate an SNMP request.

```
user@host> show snmp mib get .1.3.6.1.4.1.2636.13.61.1.9.1.1.1
juniperMIB.13.61.1.9.1.1.1 = 211
```

```
user@host> show snmp mib get .1.3.6.1.4.1.2636.13.61.1.9.1.1.2
juniperMIB.13.61.1.9.1.1.2 = 429
```

```
user@host> show snmp mib get-next .1.3.6.1.4.1.2636.13.61.1.9.1.1.1
juniperMIB.13.61.1.9.1.1.2 = 429
```

```
user@host> show snmp mib walk .1.3.6.1.4.1.2636.13.61.1.9.1.1
juniperMIB.13.61.1.9.1.1.1 = 211
juniperMIB.13.61.1.9.1.1.2 = 429
```

The system log file contains the following messages after script execution:

```
Jul  3 10:07:48  host cscript: snmp-action = get snmp-oid = .1.3.6.1.4.1.2636.13.61.1.9.1.1.1
Jul  3 10:07:51  host cscript: snmp-action = get snmp-oid = .1.3.6.1.4.1.2636.13.61.1.9.1.1.2
```

```
Jul  3 10:08:05  host cscript: snmp-action = get-next snmp-oid
= .1.3.6.1.4.1.2636.13.61.1.9.1.1.1
Jul  3 10:08:24  host cscript: snmp-action = get-next snmp-oid = .1.3.6.1.4.1.2636.13.61.1.9.1.1
Jul  3 10:08:24  host cscript: snmp-action = get-next snmp-oid
= .1.3.6.1.4.1.2636.13.61.1.9.1.1.1
Jul  3 10:08:24  host cscript: snmp-action = get-next snmp-oid
= .1.3.6.1.4.1.2636.13.61.1.9.1.1.2
```

RELATED DOCUMENTATION

SLAX Templates Overview | 97
syslog() Function (Python, SLAX, and XSLT) | 420
show snmp mib
snmp

CHAPTER 45

Troubleshoot SNMP Scripts

IN THIS CHAPTER

Trace SNMP Script Processing on Devices Running Junos OS

IN THIS SECTION

SNMP script tracing operations track SNMP 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 SNMP script tracing is to log important events, which include errors, warnings, progress messages, and script processing events, in the **/var/log/snmp.log** file on the device. When the file **snmp.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 **snmp.log.0.gz**, then **snmp.log.1.gz**, until there are 10 trace files. Then the oldest trace file (**snmp.log.9.gz**) is overwritten.

Minimum Configuration for Enabling Traceoptions for SNMP Scripts

If no SNMP script trace options are configured, the simplest way to view the trace output of an SNMP script is to configure the `output` trace flag and issue the `show log snmp.log | last` command. To do this, perform the following steps:

1. If you have not done so already, enable an SNMP script by including the `file` statement at the `[edit system scripts snmp]` hierarchy level.

```
[edit system scripts snmp]
user@host# set file filename oid oid
```

2. Enable trace options by including the `traceoptions flag output` statement at the `[edit system scripts snmp]` hierarchy level.

```
[edit system scripts snmp]
user@host# set traceoptions flag output
```

3. Commit the configuration.

```
[edit]
user@host# commit
```

4. To display the resulting trace messages recorded at the end of the **/var/log/snmp.log** file, issue the `show log snmp.log | last` operational mode command.

```
[edit]
user@host# run show log snmp.log | last
```

Table 65 on page 1141 summarizes useful filtering commands that display selected portions of the `snmp.log` file.

Table 65: SNMP Script Tracing Operational Mode Commands

Task	Command
Display logging data associated with all SNMP script processing.	`show log snmp.log`
Display processing for only the most recent operation.	`show log snmp.log | last`
Display processing for script errors.	`show log snmp.log | match error`

Table 65: SNMP Script Tracing Operational Mode Commands *(Continued)*

Task	Command
Display processing for a particular script.	`show log snmp.log | match` *`filename`*

Example: Minimum Configuration for Enabling Traceoptions for SNMP Scripts

Display the trace output of the SNMP script file **sample_snmp.slax** in the default SNMP script log file:

```
[edit]
system {
    scripts {
        snmp {
            file sample_snmp.slax {
                oid .1.3.6.1.4.1.2636.13.61.1.9.1.1;
            }
            traceoptions {
                flag output;
            }
    }
}
```

```
[edit]
user@host# commit and-quit
```

```
user@host> show snmp mib get .1.3.6.1.4.1.2636.13.61.1.9.1.1.1
juniperMIB.13.61.1.9.1.1.1 = 211
user@host> show log snmp.log | last
Mar  1 11:00:09 snmp script processing begins
Mar  1 11:00:09 running snmp script 'sample_snmp.slax'
Mar  1 11:00:09 opening snmp script '/var/run/scripts/snmp//sample_snmp.slax'
Mar  1 11:00:09 snmp script input
Mar  1 11:00:09 begin dump
<?xml version="1.0" standalone="yes"?>
<snmp-script-input>
  <snmp-action>get</snmp-action>
```

```
  <snmp-oid>.1.3.6.1.4.1.2636.13.61.1.9.1.1.1</snmp-oid>
</snmp-script-input>
Mar  1 11:00:09 end dump
Mar  1 11:00:09 reading snmp script 'sample_snmp.slax'
Mar  1 11:00:09 snmp script output
Mar  1 11:00:09 begin dump
<?xml version="1.0"?>
<snmp-script-results>
  <snmp-oid>.1.3.6.1.4.1.2636.13.61.1.9.1.1.1</snmp-oid>
  <snmp-type>Integer32</snmp-type>
  <snmp-value>211</snmp-value>
</snmp-script-results>
Mar  1 11:00:09 end dump
Mar  1 11:00:09 inspecting snmp output 'sample_snmp.slax'
Mar  1 11:00:09 sample_snmp.slax triggered for oid = .1.3.6.1.4.1.2636.13.61.1.9.1.1.1, with
type = Integer32, and with value = 211
Mar  1 11:00:09 finished snmp script 'sample_snmp.slax'
Mar  1 11:00:09 snmp script processing ends
```

Configuring Tracing of SNMP 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 snmp traceoptions]` hierarchy level:

```
[edit system scripts snmp 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 SNMP Script Log Filename

By default, the name of the file that records trace output is **snmp.log**. You can specify a different name by including the `file` statement at the `[edit system scripts snmp traceoptions]` hierarchy level.

```
[edit system scripts snmp traceoptions]
file filename;
```

Configuring the Number and Size of SNMP 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 snmp traceoptions file filename]` hierarchy level:

```
[edit system scripts snmp 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 SNMP Script Log Files

By default, access to the SNMP 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 snmp traceoptions file filename]` hierarchy level.

```
[edit system scripts snmp traceoptions file filename]
(world-readable | no-world-readable);
```

The `no-world-readable` statement restricts SNMP script log access to the owner. The `world-readable` statement enables unrestricted access to the SNMP script log file.

Configuring the SNMP Script Trace Operations

By default, the traceoptions `events` flag is turned on, regardless of the configuration settings, and only important events are logged. This includes errors, warnings, progress messages, and script processing events. You can configure the trace operations to be logged by including the following statements at the `[edit system scripts snmp traceoptions]` hierarchy level:

```
[edit system scripts snmp traceoptions]
flag all;
flag events;
flag input;
flag offline;
flag output;
flag rpc;
flag xslt;
```

Table 66 on page 1145 describes the meaning of the SNMP script tracing flags.

Table 66: SNMP Script Tracing Flags

Flag	Description	Default Setting
`all`	Trace all operations.	Off
`events`	Trace important events, including errors, warnings, progress messages, and script processing events.	On

Table 66: SNMP Script Tracing Flags *(Continued)*

Flag	Description	Default Setting
`input`	Trace SNMP script input data.	Off
`offline`	Generate data for offline development.	Off
`output`	Trace SNMP script output data.	Off
`rpc`	Trace SNMP script RPCs.	Off
`xslt`	Trace the Extensible Stylesheet Language Transformations (XSLT) library.	Off

Trace Script Processing on Devices Running Junos OS Evolved

IN THIS SECTION

When you execute an interactive script, the script can generate output, including warnings and errors, in the CLI or RPC reply. When the system triggers non-interactive scripts, for example, when an event policy triggers an event script, the script does not direct output to the terminal. In either case, you might need more information about the execution of the script. Junos OS Evolved captures trace data for all applications by default. You can view the collected traces for additional script processing information, including the memory and CPU usage, script arguments, script execution, and warnings and errors.

Junos OS Evolved collects trace data from all applications on all nodes on the Routing Engine. Whereas Junos OS logs the trace data for each type of script in separate log files, Junos OS Evolved stores the trace data for all scripts in the same location. The trace log includes data for commit, event, op, and SNMP scripts; YANG action and translation scripts; and Juniper Extension Toolkit scripts.

How to Display Trace Data for Scripts

Junos OS Evolved stores the trace data from all nodes that is collected on the primary Routing Engine under the **/var/log/traces** directory. The `cscript` application handles scripts, and the trace data for scripts is stored under the ***node*.cscript.*sequence-number*** subdirectories.

To view trace data for scripts, issue the `show trace application cscript` operational mode command.

```
user@host> show trace application cscript
2021-05-20 09:11:42.239695672 re0:cscript:4176:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg =
"Process's current softlimit [134217728] hardlimit [-1]"
2021-05-20 09:11:42.239773157 re0:cscript:4176:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg =
"Process's limits are already set by parent process"
2021-05-20 09:11:42.239812430 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"op script processing begins"
2021-05-20 09:11:42.239855140 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: /usr/libexec/ui/cscript"
2021-05-20 09:11:42.239865140 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -mop"
2021-05-20 09:11:42.239866196 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -p"
2021-05-20 09:11:42.239867156 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: /"
2021-05-20 09:11:42.239868116 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -Q2"
2021-05-20 09:11:42.239869131 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -f"
2021-05-20 09:11:42.239882048 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: hello.py"
2021-05-20 09:11:42.239883202 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -d134217728"
2021-05-20 09:11:42.239884135 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -E"
2021-05-20 09:11:42.239885131 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg:  user admin logname admin host host tty /dev/pts/0 agent op-script current-
directory /var/home/admin pid 32212 ppid 32206"
2021-05-20 09:11:42.239886175 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -u"
2021-05-20 09:11:42.239887176 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: admin"
2021-05-20 09:11:42.239888251 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -U"
```

```
2021-05-20 09:11:42.239889287 re0:cscript:4176:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
msg = "arg: -i9"
2021-05-20 09:11:42.245988806 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"running op script 'hello.py'"
2021-05-20 09:11:42.246006519 re0:cscript:4176:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg =
"opening op script '/var/db/scripts/op/hello.py'"
...
```

You can include the `terse` option to display just the timestamp and message.

```
user@host> show trace application cscript terse
2021-05-20 09:11:42.239695672 msg = "Process's current softlimit [134217728] hardlimit [-1]"
2021-05-20 09:11:42.239773157 msg = "Process's limits are already set by parent process"
2021-05-20 09:11:42.239812430 msg = "op script processing begins"
...
```

You can also refine the traces to display by specifying the trace time elapsed, process ID, and node. For example, the following command shows trace data for a specific process ID.

```
user@host> show trace application cscript pid 10683
2021-05-24 09:42:09.552687492 re0:cscript:10683:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg
= "Process's current softlimit [134217728] hardlimit [-1]"
2021-05-24 09:42:09.552819712 re0:cscript:10683:TRACE_INFO CSCRIPT_RUSAGE_CSCRIPT_RUSAGE_1  msg
= "Process's limits are already set by parent process"
2021-05-24 09:42:09.552897412 re0:cscript:10683:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1  msg
= "action script processing begins"
2021-05-24 09:42:09.553025992 re0:cscript:10683:TRACE_INFO
CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1  msg = "arg: /usr/libexec/ui/cscript"
2021-05-24 09:42:09.553095062 re0:cscript:10683:TRACE_INFO
CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1  msg = "arg: -maction"
...
```

How to Modify Trace Settings for Scripts

Junos OS Evolved traces script processing by default and traces all applications at the info level for informational messages. You can configure trace settings for specific applications at the `[edit system application]` hierarchy level. For example, you can specify the trace level of the application on a given node.

To modify script tracing operations, configure the settings under the `[edit system trace application cscript]` hierarchy level. The following example configures the `cscript` application to trace script processing on node re0 at the debug level:

```
[edit]
user@host# set system trace application cscript node re0 level debug
user@host# commit
```

For more information about configuring trace settings, see *trace*.

12
PART

Configuration Statements and Operational Commands

Junos CLI Reference Overview

We've consolidated all Junos CLI commands and configuration statements in one place. Learn about the syntax and options that make up the statements and commands and understand the contexts in which you'll use these CLI elements in your network configurations and operations.

- Junos CLI Reference

Click the links to access Junos OS and Junos OS Evolved configuration statement and command summary topics.

- Configuration Statements
- Operational Commands