JUNIPER NETWORKS

Junos® OS

Operations Automation

Release
11.4

Published: 2011-11-08
Revision 1

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

Junos® OS Operations Automation

Revision History
October 2011—Revision 1; initial release

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

END USER LICENSE AGREEMENT

The Juniper Networks product that is the subject of this technical documentation consists of (or is intended for use with) Juniper Networks software. Use of such software is subject to the terms and conditions of the End User License Agreement ("EULA") posted at http://www.juniper.net/support/eula.html. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

Table of Contents

List of Figures

List of Tables

PART 1

Overview

CHAPTER 1

Op Scripts Overview

Op Script Overview

Junos OS operation (op) scripts automate network and device management and troubleshooting. Op scripts can perform any function available through the remote procedure calls (RPCs) supported by either the Junos XML management protocol or the Junos Extensible Markup Language (XML) API. Op scripts can be executed manually in the CLI or upon user login, or they can be called from another script. They are executed by the Junos OS management (mgd) process.

Op scripts enable you to do the following things:

- 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 creating a general operation script that periodically checks network warning parameters, such as high CPU usage.

Op scripts are based on the Junos XML management protocol, and the Junos XML API, which are discussed in Junos XML API and Junos XML Management Protocol Overview. Op scripts can be written in either the Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) scripting language. Op scripts use XPath to locate the operational objects to be inspected and XSLT constructs to specify the actions to perform on the located operational objects. The actions can change the output or execute additional commands based on the output.

Related Documentation

- SLAX Overview
- XPath Overview
- XSLT Overview

How Op Scripts Work

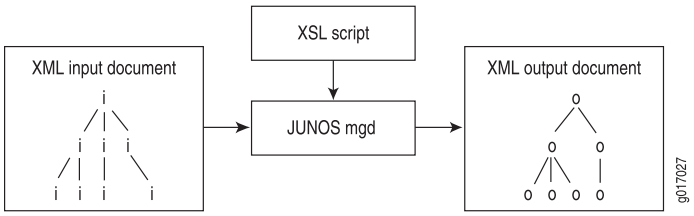
Op scripts execute Junos OS operational commands and inspect the resulting output. After inspection, op scripts can automatically correct errors within the device running Junos OS based on this output.

You add op scripts to device operations by listing the filenames of one or more op script files within the **[edit system scripts op]** hierarchy level. These files must be added to the appropriate op script file directory. For more information about op script file directories, see Storing Scripts in Flash Memory. Once added to the device, op scripts are invoked from the command line, using the **op *filename*** command.

You can use op scripts to generate changes to the device configuration by including the **<load-configuration>** tag element. Because the changes are loaded before the standard validation checks are performed, they are validated for correct syntax, just like statements already present in the configuration before the script is applied. If the syntax is correct, the configuration is activated and becomes the active, operational device configuration.

Figure 1 on page 4 shows a high-level view of the flow of op script input and output.

Figure 1: Op Script Input and Output



PART 2

Configuration

CHAPTER 2

Creating and Executing Op Scripts

Required Boilerplate for Op Scripts

When you write operation (op) scripts, you use Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) tools provided with Junos OS. These tools include basic boilerplate that you must include in all op scripts, optional extension functions that accomplish scripting tasks more easily, and named templates that make scripts easier to read and write, which you import from a file called **junos.xsl**. For more information about the extension functions and templates, see Junos Script Automation: Extension Functions in the jcs Namespace Overview and Junos Script Automation: Named Templates in the jcs Namespace Overview.

Op scripts are based on Junos XML and Junos XML protocol tag elements. Like all XML elements, angle brackets enclose the name of a Junos XML or Junos XML protocol tag element in its opening and closing tags. This is an XML convention, and the brackets are a required part of the complete tag element name. They are not to be confused with the angle brackets used in the documentation to indicate optional parts of Junos OS CLI command strings.

You must include either XSLT or SLAX boilerplate as the starting point for all op scripts that you create. The XSLT boilerplate follows:

XSLT Boilerplate for Op Scripts

```
1 <?xml version="1.0" standalone="yes"?>
2 <xsl:stylesheet version="1.0"
```

```
 3    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 4    xmlns:junos="http://xml.juniper.net/junos/*/junos"
 5    xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
 6    xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
 7    <xsl:import href="../import/junos.xsl"/>

 8    <xsl:template match="/">
 9      <op-script-results>
       <!-- ... insert your code here ... -->
10      </op-script-results>
11    </xsl:template>
    <!-- ... insert additional template definitions here ... -->
12  </xsl:stylesheet>
```

Line 1 is the Extensible Markup Language (XML) processing instruction (PI), which marks this file as XML and specifies the version of XML as 1.0. The XML PI, if present, must be the first non-comment token in the script file.

```
1  <?xml version="1.0"?>
```

Line 2 opens the style sheet and specifies the XSLT version as 1.0.

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

Lines 3 through 6 list all the namespace mappings commonly used in operation scripts. Not all of these prefixes are used in this example, but it is not an error to list namespace mappings that are not referenced. Listing all namespace mappings prevents errors if the mappings are used in later versions of the script.

```
3    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4    xmlns:junos="http://xml.juniper.net/junos/*/junos"
5    xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
6    xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
```

Line 7 is an XSLT import statement. It loads the templates and variables from the file referenced as **../import/junos.xsl**, which ships as part of Junos OS (in the file **/usr/libdata/cscript/import/junos.xsl**). The **junos.xsl** file contains a set of named templates you can call in your scripts. These named templates are discussed in Junos Script Automation: Named Templates in the jcs Namespace Overview and Junos Named Templates in the jcs Namespace Summary.

```
7    <xsl:import href="../import/junos.xsl"/>
```

Line 8 defines a template that matches the **</>** element. The **<xsl:template match="/">** element is the root element and represents the top level of the XML hierarchy. All XML Path Language (XPath) expressions in the script must start at the top level. This allows the script to access all possible Junos XML and Junos XML protocol remote procedure calls (RPCs). For more information, see XPath Overview.

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

After the **<xsl:template match="/">** tag element, the **<op-script-results>** and **</op-script-results>** container tags must be the top-level child tags, as shown in Lines 9 and 10.

```
9      <op-script-results>
       <!-- ... insert your code here ... -->
10     </op-script-results>
```

Line 11 closes the template.

```
11    </xsl:template>
```

Between Line 11 and Line 12, you can define additional XSLT templates that are called from within the **<xsl:template match="/">** template.

Line 12 closes the style sheet and the op script.

```
12  </xsl:stylesheet>
```

SLAX Boilerplate for Op Scripts

The corresponding SLAX boilerplate is as follows:

```
version 1.0;

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

match / {
  <op-script-results> {
    /*
      * Insert your code here
    */
  }
}
```

Mapping Operational Mode Commands and Output Fields to Junos XML Notation

In op scripts, you use tag elements from the Junos XML API to represent operational mode commands and output fields. For the Junos XML equivalent of commands and output fields, consult the *Junos XML API Operational Reference*.

You can also display the Junos XML tag elements for operational mode command output by directing the output from the command to the **| display xml** command:

```
user@host> command-string | display xml
```

For example:

```
user@host> show interfaces terse | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/junos">
    <interface-information
xmlns="http://xml.juniper.net/junos/10.0RI0/junos-interface" junos:style="terse">
        <physical-interface>
            <name>dsc</name>
            <admin-status>up</admin-status>
            <oper-status>up</oper-status>
        </physical-interface>
        <physical-interface>
            <name>fxp0</name>
            <admin-status>up</admin-status>
            <oper-status>up</oper-status>
            <logical-interface>
                <name>fxp0.0</name>
                <admin-status>up</admin-status>
```

```
                    <oper-status>up</oper-status>
                    ...
```

Related Documentation

- Configuring Help Text for Op Scripts on page 15
- Declaring Arguments in Op Scripts on page 13
- Using RPCs and Operational Mode Commands in Op Scripts on page 10

Using RPCs and Operational Mode Commands in Op Scripts

Most Junos OS operational mode commands have XML equivalents. These XML commands can be executed remotely using the *remote procedure call* (RPC) protocol. All operational mode commands that have XML equivalents are listed in the *Junos XML API Operational Reference*.

Use of RPC and operational mode commands in op scripts is discussed in more detail in the following sections:

- Using RPCs in Op Scripts on page 10
- Displaying the RPC Tags for a Command on page 11
- Using Operational Mode Commands in Op Scripts on page 12

Using RPCs in Op Scripts

To use an RPC in an op script, include the RPC in a variable declaration. You then invoke the RPC with the **jcs:invoke()** or **jcs:execute()** extension function and include the RPC variable as an argument. The **jcs:invoke()** function executes the RPC on the local device. You can use the **jcs:execute()** function with a connection handle to execute the RPC on a remote device.

The following snippet, which invokes an RPC on the local device, is expanded and fully described in "Example: Customizing Output of the show interfaces terse Command Using an Op Script" on page 28.

XSLT Syntax

```
<xsl:variable name="rpc">
  <get-interface-information/> # Junos RPC for the show interfaces command
</xsl:variable>
<xsl:variable name="out" select="jcs:invoke($rpc)"/>
...
```

SLAX Syntax

```
var $rpc = <get-interface-information>;
var $out = jcs:invoke($rpc);
```

To execute an RPC on a remote device, an SSH session must be established. In order for the script to establish the connection, you must either configure the SSH host key information for the remote device on the local device where the script will be executed, or the SSH host key information for the remote device must exist in the known hosts file of the user executing the script. For each remote device where an RPC is executed, configure the SSH host key information with one of the following methods:

- To configure SSH known hosts on the local device, include the **host** statement, and specify hostname and host key options for the remote device at the **[edit security ssh-known-hosts]** hierarchy level of the configuration.
- To manually retrieve SSH host key information, issue the **set security ssh-known-hosts fetch-from-server *hostname*** configuration mode command to instruct Junos OS to connect to the remote device and add the key.

```
user@host# set security ssh-known-hosts fetch-from-server router2

The authenticity of host 'router2 (10.10.10.1)' can't be established.
RSA key fingerprint is 30:18:99:7a:3c:ed:40:04:0f:fd:c1:57:7e:6b:f3:90.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'router2,10.10.10.1' (RSA) to the list of known
 hosts.
```

- To manually import SSH host key information from a file, use the **set security ssh-known-hosts load-key-file *filename*** configuration mode command and specify the known-hosts file.

```
user@host# set security ssh-known-hosts load-key-file /var/tmp/known_hosts

Import SSH host keys from trusted source /var/tmp/known_hosts ? [yes,no]
 (no) yes
```

- Alternatively, the user executing the script can log in to the local device, SSH to the remote device, and then manually accept the host key, which is added to that user's known hosts file. In the following example, root is logged in to **router1**. In order to execute a remote RPC on **router2**, root adds the host key of **router2** by issuing the **ssh router2** operational mode command and manually accepting the key.

```
root@router1> ssh router2

The authenticity of host 'router2 (10.10.10.1)' can't be established.
RSA key fingerprint is 30:18:99:7a:3c:ed:40:04:0f:fd:c1:57:7e:6b:f3:90.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'router2,10.10.10.1' (RSA) to the list of known
 hosts.
```

Displaying the RPC Tags for a Command

To display the remote procedure call (RPC) XML tags for an operational mode command, enter **display xml rpc** after the pipe symbol (|).

The following example displays the RPC tags for the **show route** command:

```
user@host> show route | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.1I0/junos">
    <rpc>
        <get-route-information>
        </get-route-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Using Operational Mode Commands in Op Scripts

Some operational mode commands do not have XML equivalents. If a command is not listed in the *Junos XML API Operational Reference*, the command does not have an XML equivalent.

Another way to determine whether a command has an XML equivalent is to issue the command followed by the **| display xml** command:

user@host> ***operational-mode-command*** **| display xml**

If the output includes only tag elements like **<output>**, **<cli>**, and **<banner>**, the command might not have an XML equivalent. In the following example, the output indicates that the **show host** command has no XML equivalent:

```
user@host> show host hostname | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/junos">
    <output>
       ...
    </output>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

NOTE: For some commands that have an XML equivalent, the output of the piped **| display xml** command does not include tag elements other than **<output>**, **<cli>**, and **<banner>** only because the relevant feature is not configured. For example, the **show services cos statistics forwarding-class** command has an XML equivalent that returns output in the **<service-cos-forwarding-class-statistics>** response tag, but if the configuration does not include include any statements at the **[edit class-of-service]** hierarchy level then there is no actual data for the **show services cos statistics forwarding-class | display xml** command to display. The output is something like this:

```
user@host> show services cos statistics forwarding-class | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/8.3IO/junos">

    <cli>

        <banner></banner>

    </cli>

</rpc-reply>
```

For this reason, the information in the *Junos XML API Operational Reference* is normally more reliable.

An op script can include commands that have no XML equivalent. Use the **<command>**, **<xsl:value-of>**, and **<output>** elements in the script, as shown in the following code snippet. This snippet is expanded and fully described in "Example: Displaying DNS Hostname Information Using an Op Script" on page 39.

```
<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>
...
```

Related Documentation

- Configuring Help Text for Op Scripts on page 15
- Declaring Arguments in Op Scripts on page 13
- Mapping Operational Mode Commands and Output Fields to Junos XML Notation on page 9

Declaring Arguments in Op Scripts

There are two ways to declare arguments to an op script: by including XSLT or SLAX instructions in the script or by including statements in the Junos configuration. *Script-generated* and *configuration-generated* arguments have the same operational impact.

To declare arguments within a script, declare a global variable named **arguments**, containing **<argument>** tag elements. Within each **<argument>** tag element, include the required **<name>** tag element and the optional **<description>** tag element:

XSLT Syntax

```
<xsl:variable name="arguments">
  <argument>
    <name>name</name>
    <description>name</description>
  </argument>
</xsl:variable>
```

SLAX Syntax

```
var $arguments = {
  <argument> {
    <name> "name";
    <description> "descriptive-text";
  }
}
```

To declare arguments in the configuration, include the **arguments** statement in the **[edit system scripts op file *filename*]** hierarchy level.

```
[edit system scripts op file filename]
arguments {
```

```
    argument-name {
      description descriptive-text;
    }
  }
```

If you include the optional **<description>** tag element or the **description** statement, the text of the description appears in the command-line interface (CLI) as a help-text string to describe the purpose of the argument, as discussed in "Configuring Help Text for Op Scripts" on page 15.

In the operation script, you must include a corresponding parameter declaration for each argument. The parameter name must match the name of the argument:

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

The SLAX equivalent is:

```
param $name;
```

You can create a hidden argument by including the **<xsl:param name="*name*"/>** instruction without listing the argument in the **arguments** variable or in the configuration.

After you declare an argument, you can use command completion to list available arguments:

```
user@host> op filename ?
Possible completions:
  argument-name        description
  argument-name        description
```

For each argument you include on the command-line, you must specify a corresponding value. To do this, include an ***argument-name*** and an ***argument-value*** when you execute the script with the **op *filename*** command:

```
user@host> op filename argument-name argument-value
```

NOTE: If you specify an argument that the script does not recognize, the script ignores the argument.

If you configure arguments by including the **arguments** statement in the configuration, any arguments that you declare directly in the script are still available, but are not listed among the **Possible completions** when you issue the **op *filename* ?** command.

If you declare all arguments in the script (and none in the configuration), then the arguments do appear in the **Possible completions** list. This is because the management (mgd) process populates the **Possible completions** list by first checking the configuration for arguments. The mgd process looks in the script for arguments only if no arguments are found in the configuration. Thus, if arguments are declared in the configuration, the arguments declared in the script become hidden in the CLI.

Example: Declaring Arguments

Declare two arguments named **interface** and **protocol**. Execute the script, specifying the **ge-0/2/0.0** interface and the **inet** protocol as values for the arguments. For either method, you must declare corresponding script parameters:

```
<xsl:param name="interface"/>
<xsl:param name="protocol"/>
```

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)";
    }
  }
}
```

Executing the Script

```
user@host> op script1 interface ge-0/2/0.0 protocol inet
```

Related Documentation

- Example: Importing Files Using an Op Script on page 53
- Configuring Help Text for Op Scripts on page 15
- Mapping Operational Mode Commands and Output Fields to Junos XML Notation on page 9
- Using RPCs and Operational Mode Commands in Op Scripts on page 10

Configuring Help Text for Op Scripts

You can provide help text to describe an op script and its arguments when the **?** is used to list possible completions in the CLI. Include the **description** statement:

```
description descriptive-text;
```

You can include this statement at the following hierarchy levels:

- **[edit system** scripts op file *filename***]**
- **[edit system** scripts op file *filename* arguments *argument-name***]**

The following examples show the configuration and the resulting output.

Examples: Configuring Help Text for Op Scripts

Configure help text for a script and display the resulting output:

```
[edit system scripts op]
user@host# set file interface.xsl description "Test the interface"
user@host# commit
...
[edit system scripts op]
user@host# set file ?
Possible completions:
  <name>          Local filename of the script file
  interface.xsl   Test the interface
```

Configure help text for a script's arguments and display the resulting output:

```
[edit system scripts op file interface.xsl arguments]
user@host# set t1 description "Search for T1 interfaces"
user@host# set t3 description "Search for T3 interfaces"
user@host# commit
...
[edit system scripts op file interface.xsl arguments]
user@host# set ?
Possible completions:
  <name>          Name of the argument
  t1              Search for T1 interfaces
  t3              Search for T3 interfaces
```

Related Documentation

- Declaring Arguments in Op Scripts on page 13
- Mapping Operational Mode Commands and Output Fields to Junos XML Notation on page 9
- Using RPCs and Operational Mode Commands in Op Scripts on page 10

Enabling an Op Script and Defining a Script Alias

Operation (op) scripts are stored on a device's hard drive in the **/var/db/scripts/op** directory or on the flash drive in the **/config/scripts/op** directory. Only users in the Junos OS **super-user** login class can access and edit files in these directories. For information about setting the storage location for scripts, see Storing Scripts in Flash Memory.

NOTE: If the device has dual Routing Engines and you want to enable an op script to execute on both Routing Engines, you must copy the script to the **/var/db/scripts/op** or **/config/scripts/op** directory on both Routing Engines. The **commit synchronize** command does not automatically copy scripts between Routing Engines.

You must enable an op script before it can be executed. Include the **file *filename*** statement at the **[edit system scripts op]** hierarchy level, specifying the name of an XSLT or SLAX file containing an op script. Only users who belong to the Junos **super-user** login class can enable op scripts.

```
[edit system scripts op]
file filename;
```

The filename of an op script written in SLAX must include the **.slax** extension for the script to be enabled and executed. No particular filename extension is required for op scripts written in XSLT, but we strongly recommend that you append the **.xsl** extension. Whether or not you choose to include the **.xsl** extension on the file, the filename that you add at the **[edit system scripts op]** hierarchy level must exactly match the filename of the script in the directory. For example, if the XSLT script filename is **script1.xsl**, then you must include **script1.xsl** in the configuration hierarchy to enable the script; likewise, if the XSLT script filename is **script1**, then you must include **script1** in the configuration hierarchy.

To determine which op scripts are currently enabled on the device, use the **show** command to display the files included at the **[edit system scripts op]** hierarchy level. 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.

Optionally, you can define an alias for an op script and then specify either the filename or the alias when you execute the script. To define the alias, include the **command** statement at the **[edit system scripts op file *filename*]** hierarchy level:

```
[edit system scripts op]
file filename {
    command filename-alias;
}
```

Executing an Op Script

Unlike commit scripts, operation (op) scripts do not execute during a commit operation. When you issue the **commit** command, op scripts configured at the **[edit system scripts op]** hierarchy level are placed into system memory and enabled for execution. After the commit operation completes, you can execute an op script from the CLI by issuing the **op** operational mode command. You also can configure the device to execute an op script automatically when a member of a specified Junos OS login class logs in to the CLI.

Executing an Op Script by Issuing the op Command

To execute an op script from the CLI, issue the **op** operational mode command, specifying a URL, the script filename, or the alias defined by the **command** statement at the **[edit system scripts op file *filename*]** hierarchy level.

```
user@host> op (filename-or-alias | url url)
```

Executing an Op Script at Login

You can configure an op script to execute automatically when any user belonging to a designated Junos OS login class logs in to the CLI. To associate an op script with a login class, include the **login-script *script-filename*** statement at the **[edit system login class *class-name*]** hierarchy level:

```
[edit system login]
class class-name {
   login-script script-filename;
}
```

The following example configures the **super-user-login.slax** op script to execute when any user who belongs to the **super-user** class logs in to the CLI (provided that the script has been enabled as discussed in "Enabling an Op Script and Defining a Script Alias" on page 16).

```
[edit system login]
class super-user {
   login-script super-user-login.slax;
}
```

Configuring Checksum Hashes for an Op Script

You can configure one or more checksum hashes that can be used to verify the integrity of an op script before the script runs on the switch, router, or security device.

To configure a checksum hash:

1. Create the script.
2. Place the script in the **/var/db/scripts/op** directory on the device.
3. Run the script through one or more hash functions to calculate hash values.

   Junos OS supports MD5, SHA-1, and SHA-256 hash functions.

   ```
   user@host> file checksum md5 /var/db/scripts/commit/script1.slax
     MD5 (/var/db/scripts/op/script1.slax) = 3af7884eb56e2d4489c2e49b26a39a97
   user@host> file checksum sha1 /var/db/scripts/commit/script1.slax
     SHA1 (/var/db/scripts/op/script1.slax) =
     00dc690fb08fb049577d012486c9a6dad34212c0
   user@host> file checksum sha-256 /var/db/scripts/commit/script1.slax
     SHA256 (/var/db/scripts/op/script1.slax) =
     150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
   ```

4. Configure the script with one or more hash values.

   ```
   [edit system scripts op]
   user@host# set file script1.slax checksum md5 3af7884eb56e2d4489c2e49b26a39a97
   [edit system scripts op]
   user@host# set file script1.slax checksum
     sha-1 00dc690fb08fb049577d012486c9a6dad34212c0
   [edit system scripts op]
   user@host# set file script1.slax checksum
     sha-256 150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
   ```

During the execution of the script, Junos OS recalculates the checksum value using the configured hash and verifies that the calculated value matches the configured value. If the values differ, the execution of the script fails. When you configure multiple checksum values with different hash algorithms, all the configured values must match the calculated values; otherwise, the script execution fails.

NOTE: If the op script is stored remotely, do not include the **checksum** statement in the configuration. You can verify the script's integrity before it runs by specifying the hash value on the command line when you run the **op** command with the <url> option and the <key> option.

Related Documentation

- Configuring Checksum Hashes for a Commit Script
- Configuring Checksum Hashes for an Event Script
- file checksum md5 command in the *System Basics and Services Command Reference*
- file checksum sha-256 command in the *System Basics and Services Command Reference*
- file checksum sha1 command in the *System Basics and Services Command Reference*
- op command in the *System Basics and Services Command Reference*

Executing an Op Script from a Remote Site

As an alternative to storing operation (op) scripts locally on the device, you can store op scripts at a remote site. This allows you to execute the scripts by specifying a URL on the command line.

To execute an op script from a remote site:

1. Create the script.
2. (Optional) Store the script temporarily in the **/var/tmp** directory on the device, and run the script through one or more hash functions to calculate hash values.

   Junos OS supports MD5, SHA-1, and SHA-256 hash functions.

   ```
   user@host> file checksum md5 /var/tmp/script1.slax
     MD5 (/var/tmp/script1.slax) = 3af7884eb56e2d4489c2e49b26a39a97
   user@host> file checksum sha1 /var/tmp/script1.slax
     SHA1 (/var/tmp/script1.slax) = 00dc690fb08fb049577d012486c9a6dad34212c0
   user@host> file checksum sha-256 /var/tmp/script1.slax
     SHA256 (/var/tmp/script1.slax) =
     150bf53383769f3bfedd41fe73320777f208d4fda81230cb27b8738
   ```

3. Place the script on the remote server.
4. Provide the script URL and the optional hash values to the administrators who will execute the script.
5. Execute the script by running the **op** command and specifying the URL that points to the remote file.

```
user@host> op url https://www.juniper.net/scripts/2009-04-01.01.slax
  key md5 3af7884eb56e2d4489c2e49b26a39a97
```

This example shows how to include the <key> option and the MD5 checksum information.

NOTE: If the op script is stored locally, do not include the hash key on the command line. Instead, configure the hash value by including the **checksum** statement at the **[edit system scripts op file *filename*]** hierarchy level. During the execution of the script, Junos OS recalculates the checksum value using the configured hash and verifies that the calculated value matches the configured value.

To prevent the execution of 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
```

When you configure the **no-allow-url** statement, issuing the **op url *url*** operational mode command generates an error.

Related Documentation

- Configuring Checksum Hashes for an Op Script on page 18
- file checksum md5 command in the *System Basics and Services Command Reference*
- file checksum sha-256 command in the *System Basics and Services Command Reference*
- file checksum sha1 command in the *System Basics and Services Command Reference*
- no-allow-url on page 72
- op command in the *System Basics and Services Command Reference*

Disabling 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, perform the following steps:

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, perform the following steps:

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 drive 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

- Enabling an Op Script and Defining a Script Alias on page 16

CHAPTER 3

Op Script Examples

Example: Changing the Configuration Using an Op Script

This example explains how to make structured changes to the Junos OS configuration using an op script.

Requirements

This example uses a device running Junos OS.

Overview and Op Script

Op scripts can be used to make structured changes to the Junos OS configuration using the **jcs:load-configuration** template, which is included in the import file **junos.xsl**. Experienced users, who are familiar with Junos OS, can write scripts that prompt for the relevant configuration information and modify the configuration accordingly. This allows users who have less experience with Junos OS to safely modify the configuration using the script.

When called, the **jcs:load-configuration** template performs the following actions:

1. Locks the configuration database
2. Loads the configuration changes
3. Commits the configuration
4. Unlocks the configuration database

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

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

You provide arguments to the **jcs:load-configuration** template to specify how to integrate the changes into the existing configuration, how to customize the commit operation, what changes to make to the configuration, and which connection handle to use. The XSLT and SLAX syntax for the template call is:

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

```
call jcs:load-configuration($action="(merge | override | replace)",
 $commit-options=node-set, $configuration=configuration-data,
 $connection=connection-handle);
```

The following sample SLAX script demonstrates how to use the **jcs:load-configuration** template to disable an interface on a device running Junos OS. All of the values required for the **jcs:load-configuration** template are defined as variables, which are then passed into the template as arguments.

In this example, the **$usage** variable is initialized with a general description of the function of the script. When the script is run, the usage description is output to the CLI using a call to the **jcs:output()** function. This allows the user to verify that he is using the script for the correct purpose.

The script calls the **jcs:get-input()** function and prompts the user to enter the name of the interface that should be disabled. The interface name is stored in the **$interface** variable. The configuration data that includes the changes to the configuration are stored in the variable **$config-changes**. This is the value used for the **$configuration** parameter of the **jcs:load-configuration** template. This variable includes the Junos XML API tags for the configuration statements that are to be modified. The variable **$interface**, which is supplied by the user, designates the name of the interface to disable.

The **$load-action** variable is initialized to **merge**, which merges the configuration changes in the **$disable** variable with the candidate configuration. This is the equivalent of the CLI configuration mode command **load merge**. Other load options include **replace** and **override**.

The **$options** variable uses the **:=** operator to create a node-set, which is passed to the template as the value of the **$commit-options** parameter. This example includes the **log** tag to add the description of the commit to the commit log file for future reference.

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

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";

import "../import/junos.xsl";

match / {
<op-script-results> {

  var $usage = "This script disables the interface specified by the user." _
               "The script modifies the candidate configuration to disable " _
               "the interface and commits the configuration to activate it.";
  var $temp = jcs:output($usage);

  var $interface = jcs:get-input("Enter interface to disable: ");

  var $config-changes = {
    <configuration> {
      <interfaces> {
        <interface> {
          <name> $interface;
          <disable>;
        }
      }
    }
  }

  var $load-action = "merge";

  var $options := {
    <commit-options> {
      <log> "disabling interface " _ $interface;
    }
  }

  var $conn_handle = jcs:open();

  var $results := {
   call jcs:load-configuration( $action=$load-action, $commit-options=$options,
```

```
            $configuration=$config-changes, $connection=$conn_handle);
    }

    $close-results = jcs:close($conn_handle);
  }
}
```

The **:=** operator copies the results of the **jcs:load-configuration** template call to a temporary variable and runs the **node-set** function on that variable. The resulting node-set is then stored in the **$results** variable. The **:=** operator ensures that the **$results** variable is a node-set rather than a result tree fragment so that the script can access the contents. The **jcs:close()** function closes the connection.

By default, the **jcs:load-configuration** template does not output messages to the CLI. To quickly view any issues with the commit, you should add code to the script to output any error or warning messages that are generated as a result of the **jcs:load-configuration** template call:

```
if ($results//xnm:error) {
  for-each ($results//xnm:error) {
    <output> message;
  }
}
if ($results//xnm:warning) {
  for-each ($results//xnm:warning) {
    <output> message;
  }
}
```

Device Configuration

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, include the **file** statement at the **[edit system scripts op]** hierarchy level and **config-change.slax**.

   ```
   [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 by issuing the **op config-change** operational mode command.

   ```
   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
```

Verification

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> file show /var/log/messages | last

Sep 22 17:08:13  host file[7319]: UI_COMMIT: User 'user' requested 'commit'
 operation (comment: disabling interface so-0/0/0)
Sep 22 17:08:16  host xntpd[1386]: ntpd exiting on signal 1
Sep 22 17:08:16  host xntpd[1386]: ntpd 4.2.0-a Fri Jun 25 13:48:13 UTC
2010 (1)
Sep 22 17:08:16  host mib2d[1434]: SNMP_TRAP_LINK_DOWN: ifIndex 526,
ifAdminStatus down(2), ifOperStatus down(2), ifName so-0/0/0
```

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

- Storing and Enabling Scripts
- jcs:close() Function
- jcs:get-input() Function
- jcs:load-configuration Template
- jcs:open() Function
- jcs:output() Function

Example: Customizing Output of the show interfaces terse Command Using an Op Script

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
                                            fec0::10:0:0:4/64
                                   tnp      4
gre                     up    up
ipip                    up    up
lo0                     up    up
lo0.0                   up    up   inet     127.0.0.1           --> 0/0
lo0.16385               up    up   inet
                                   inet6    fe80::2a0:a5ff:fe12:2f04
lsi                     up    up
mtun                    up    up
pimd                    up    up
pime                    up    up
tap                     up    up
```

In Junos XML, the output fields are represented as follows:

```
user@host> show interfaces terse | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.0R1/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.

```
1  <?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>
      <admin-status><xsl:value-of select="$status"/></admin-status>
      <xsl:choose>
        <xsl:when test="$protocol">
          <xsl:copy-of
          select="address-family[address-family-name = $protocol]"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:copy-of select="address-family"/>
        </xsl:otherwise>
      </xsl:choose>
    </logical-interface>
  </xsl:template>
</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 7.

```
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

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

Verifying the Commit 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
                                            fec0::10:0:0:4/64
                                   tnp      4
gre                     up    up
ipip                    up    up
lo0                     up    up
lo0.0                   up    up   inet     127.0.0.1           --> 0/0
lo0.16385               up    up   inet
                                   inet6    fe80::2a0:a5ff:fe12:2f04
lsi                     up    up
mtun                    up    up
pimd                    up    up
pime                    up    up
tap                     up    up


user@host> op interface
Interface               Admin Link Proto    Local                 Remote
fxp0.0                  This is the Ethernet Management interface.
                                   inet     192.168.71.246/21
fxp1.0                             inet     10.0.0.4/8
                                   inet6    fe80::200:ff:fe00:4/64
                                            fec0::10:0:0:4/64
                                   tnp      4
lo0.0                              inet     127.0.0.1           --> 0/0
lo0.16385                          inet
                                   inet6    fe80::2a0:a5ff:fe12:2f04-->
```

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: Displaying DNS Hostname Information Using an Op Script

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

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

Verifying the Commit 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: Exporting Files Using an Op Script

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

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

- Verifying the Op Script Arguments on page 47
- Verifying Op Script Execution on page 48

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

- Declaring Arguments in Op Scripts on page 13
- Example: Importing Files Using an Op Script on page 53

Example: Finding LSPs to Multiple Destinations Using an Op Script

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

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

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: Importing Files Using an Op Script

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

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

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

- Declaring Arguments in Op Scripts on page 13
- Example: Exporting Files Using an Op Script on page 43

Example: Restarting an FPC Using an Op Script

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

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

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: Searching Files Using an Op Script

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

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

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
- jcs:grep Template
- jcs:regex() Function

CHAPTER 4

Summary of Operations Automation Configuration Statements

arguments

Syntax

```
arguments {
  argument-name {
    description descriptive-text;
  }
}
```

Hierarchy Level [edit system scripts op file *filename*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, configure command-line arguments to the script.

Options ***argument-name***—The name of a command-line argument to an op script.

The remaining statement is explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Declaring Arguments in Op Scripts on page 13

checksum

Syntax checksum (md5 | sha-256 | sha1) *hash*;

Hierarchy Level [edit event-options event-script file *filename*],
[edit system scripts commit file *filename*],
[edit system scripts op file *filename*]

Release Information Statement introduced in Junos OS Release 9.5.

Description For Junos OS commit scripts and op scripts, specify the MD5, SHA-1, or SHA-256 checksum hash. When it executes a local event, commit, or op script, Junos OS verifies the authenticity of the script by using the configured checksum hash.

Options **md5 *hash***—MD5 checksum of this script.

sha-256 *hash*—SHA-256 checksum of this script.

sha1 *hash*—SHA-1 checksum of this script.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring Checksum Hashes for a Commit Script
- Configuring Checksum Hashes for an Event Script
- Configuring Checksum Hashes for an Op Script on page 18
- Executing an Op Script from a Remote Site on page 19
- file checksum md5 command in the *System Basics and Services Command Reference*
- file checksum sha-256 command in the *System Basics and Services Command Reference*
- file checksum sha1 command in the *System Basics and Services Command Reference*

command

Syntax command *filename-alias*;

Hierarchy Level [edit system scripts op file *filename*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, configure a filename alias for the script file. This allows you to run the script by referencing either the script filename or the filename alias.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Enabling an Op Script and Defining a Script Alias on page 16

description

Syntax description *descriptive-text*;

Hierarchy Level [edit system scripts op file *filename*]
[edit system scripts op file *filename* arguments *argument-name*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, provide a help-text string that appears in the command-line interface (CLI).

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Configuring Help Text for Op Scripts on page 15
- Declaring Arguments in Op Scripts on page 13
- **file (Op Scripts) on page 70**

file (Op Scripts)

Syntax

```
file filename {
  arguments {
    argument-name {
      description descriptive-text;
    }
  }
  checksum (md5 | sha-256 | sha1) hash;
  command filename-alias;
  description descriptive-text;
  refresh;
  refresh-from url;
  source url;
}
```

Hierarchy Level [edit system scripts op]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, enable an op script that is located in the **/var/db/scripts/op** directory.

Options ***filename***—The name of an Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX) file containing an op script.

The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Enabling an Op Script and Defining a Script Alias on page 16

op

Syntax

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

Hierarchy Level [edit system scripts]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, configure an operation scripting mechanism.

Options The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Storing and Enabling Scripts

no-allow-url

Syntax no-allow-url;

Hierarchy Level [edit system scripts op]

Release Information Statement introduced in Junos OS Release 10.0.

Description For Junos OS op scripts, prohibit the remote execution of scripts. When you include this configuration statement, the **op url** operational mode command generates an error and does not permit you to execute the op script from a remote site.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- file (Op Scripts) on page 70
- Executing an Op Script from a Remote Site on page 19

refresh (Op Scripts)

Syntax refresh;

Hierarchy Level [edit system scripts op],
[edit system scripts op file *filename*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, overwrite the local copy of all enabled op scripts or a single enabled script located in the **/var/db/scripts/op** directory with the copy located at the source URL, specified in the **source** statement at the same hierarchy level.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Using a Master Source Location for a Script
- refresh-from (Op Scripts) on page 73
- source (Op Scripts) on page 75

refresh-from (Op Scripts)

Syntax refresh-from *url*;

Hierarchy Level [edit system scripts op],
[edit system scripts op file *filename*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, overwrite the local copy of all enabled op scripts or a single enabled script located in the **/var/db/scripts/op** directory with the copy located at a URL other than the URL specified in the **source** statement.

Options *url*—Source specified as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Using an Alternate Source Location for a Script
- refresh (Op Scripts) on page 72
- source (Op Scripts) on page 75

scripts

Syntax

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

Hierarchy Level [edit system]

Release Information Statement introduced in Junos OS Release 7.4.

Description For Junos OS commit or op scripts, configure scripting mechanisms.

Options The statements are explained separately.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Storing and Enabling Scripts

source (Op Scripts)

Syntax source *url*;

Hierarchy Level [edit system scripts op file *filename*]

Release Information Statement introduced in Junos OS Release 7.6.

Description For Junos OS op scripts, specify the location of the source file for an enabled script located in the **/var/db/scripts/op** directory. When you include the **refresh** statement at the same hierarchy level, the local copy is overwritten by the version stored at the specified URL.

Options *url*—Master source file for an op script specified as an HTTP URL, FTP URL, or scp-style remote file specification.

Required Privilege Level maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation
- Using a Master Source Location for a Script
- refresh (Op Scripts) on page 72
- refresh-from (Op Scripts) on page 73

traceoptions (Commit and Op Scripts)

Syntax

```
traceoptions {
  file <filename> <files number> <size size> <world-readable | no-world-readable>;
  flag flag;
  no-remote-trace;
}
```

Hierarchy Level

[edit system scripts commit],
[edit system scripts op]

Release Information

Statement introduced in Junos OS Release 7.4.
Statement introduced in Junos OS Release 9.0 for EX Series switches.

Description

Define tracing operations for commit or op scripts.

Default

If you do not include this statement, no script-specific tracing operations are performed.

Options

filename—Name of the file to receive the output of the tracing operation. All files are placed in the directory **/var/log**. By default, commit script process tracing output is placed in the file **cscript.log** and op script process tracing is placed in the file **op-script.log**. If you include the **file** statement, you must specify a filename. To retain the default, you can specify **cscript.log** or **op-script.log** as the filename.

files ***number***—(Optional) Maximum number of trace files. When a trace file named ***trace-file*** reaches its maximum size, it is renamed and compressed to ***trace-file*.0.gz**. When ***trace-file*** again reaches its maximum size, ***trace-file*.0.gz** is renamed ***trace-file*.1.gz** and ***trace-file*** is renamed and compressed to ***trace-file*.0.gz**. This renaming scheme continues until the maximum number of trace files is reached. Then the oldest trace file is overwritten.

If you specify a maximum number of files, you also must specify a maximum file size with the **size** option and a filename.

Range: 2 through 1000

Default: 10 files

flag—Tracing operation to perform. To specify more than one tracing operation, include multiple **flag** statements. You can include the following flags:

- **all**—Log all operations
- **events**—Log important events
- **input**—Log script input data
- **offline**—Generate data for offline development
- **output**—Log script output data
- **rpc**—Log script RPCs
- **xslt**—Log the XSLT library

no-world-readable—Restrict file access to owner. This is the default.

size *size*—(Optional) Maximum size of each trace file, in kilobytes (KB), megabytes (MB), or gigabytes (GB). When a trace file named ***trace-file*** reaches this size, it is renamed and compressed to ***trace-file*.0.gz**. When ***trace-file*** again reaches its maximum size, ***trace-file*.0.gz** is renamed ***trace-file*.1.gz** and ***trace-file*** is renamed and compressed to ***trace-file*.0.gz**. This renaming scheme continues until the maximum number of trace files is reached. Then the oldest trace file is overwritten.

If you specify a maximum file size, you also must specify a maximum number of trace files with the **files** option and a filename.

Syntax: ***x*k** to specify KB, ***x*m** to specify MB, or ***x*g** to specify GB

Range: 10 KB through 1 GB

Default: 128 KB

world-readable—Enable unrestricted file access.

Required Privilege Level

maintenance—To view this statement in the configuration.
maintenance-control—To add this statement to the configuration.

Related Documentation

- Tracing Commit Script Processing
- Tracing Op Script Processing on page 85

PART 3

Administration

CHAPTER 5

Configuration and Operations Configuration Statements

Any Hierarchy Level

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

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

[edit system scripts] Hierarchy Level

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

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

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

PART 4

Troubleshooting

CHAPTER 6

Troubleshooting Op Scripts

Tracing Op Script Processing

Op script tracing operations track all op script operations and record them in a log file. The logged error descriptions provide detailed information to help you solve problems faster.

The default operation of op script tracing is to log important events in a file called **op-script.log** located in the **/var/log** directory. When the file **op-script.log** reaches 128 kilobytes (KB), it is renamed with a number 0 through 9 (in ascending order) appended to the end of the file and then compressed. The resulting files are **op-script.log.0.gz**, then **op-script.log.1.gz**, until there are 10 trace files. Then the oldest trace file (**op-script.log.9.gz**) is overwritten. (For more information about how log files are created, see the *Junos OS System Log Messages Reference*.)

This section discusses the following topics:

Minimum Configuration for Enabling Traceoptions for Op Scripts

If no op script trace options are configured, the simplest way to view the trace output of an op script is to configure the **output** trace flag and issue the **show log op-script.log | last** command. To do this, perform the following steps:

1. If you have not done so already, enable an op script by including the **file** statement at the **[edit system scripts op]** hierarchy level:

   ```
   [edit system scripts op]
   user@host# set file filename
   ```

2. Enable trace options by including the **traceoptions flag output** statement at the **[edit system scripts op]** hierarchy level:

   ```
   [edit system scripts op]
   user@host# set traceoptions flag output
   ```

3. Issue the **commit** command:

```
[edit]
user@host# commit
```

4. Display the resulting trace messages recorded in the file **/var/log/op-script.log** file. At the end of the log is the output generated by the op script you enabled in Step 1. To display the end of the log, issue the **show log op-script.log | last** operational mode command:

```
[edit]
user@host# run show log op-script.log | last
```

Table 1 on page 86 summarizes useful filtering commands that display selected portions of the **op-script.log** file.

Table 1: Op Script Tracing Operational Mode Commands

Task	Command
Display logging data associated with all op script processing.	**show log op-script.log**
Display processing for only the most recent operation.	**show log op-script.log | last**
Display processing for script errors.	**show log op-script.log | match error**
Display processing for a particular script.	**show log op-script.log | match *filename***

Example: Minimum Configuration for Enabling Traceoptions for Op Scripts

Display the trace output of the op script file **source-route.xsl**:

```
[edit]
system {
  scripts {
    op {
      file source-route.xsl;
      traceoptions flag output;
    }
  }
}
```

```
[edit]
user@host# commit
[edit]
user@host# run show log op-script.log | last
```

Configuring Tracing of Op Scripts

You cannot change the directory (**/var/log**) to which trace files are written. However, you can customize other trace file settings by including the following statements at the **[edit system scripts op traceoptions]** hierarchy level:

```
[edit system scripts op traceoptions]
```

```
file <filename> <files number> <size size> <world-readable | no-world-readable>;
flag all;
flag events;
flag input;
flag offline;
flag output;
flag rpc;
flag xslt;
no-remote-trace;
```

These statements are described in the following sections:

Configuring the Op Script Log Filename

By default, the name of the file that records trace output is **op-script.log**. You can specify a different name by including the **file** statement at the **[edit system scripts op traceoptions]** hierarchy level:

```
[edit system scripts op traceoptions]
file filename;
```

Configuring the Number and Size of Op Script Log Files

By default, when the trace file reaches 128 KB in size, it is renamed and compressed to ***filename*.0.gz**, then ***filename*.1.gz**, and so on, until there are 10 trace files. Then the oldest trace file (***filename*.9.gz**) is overwritten.

You can configure the limits on the number and size of trace files by including the following statements at the **[edit system scripts op traceoptions file <*filename*>]** hierarchy level:

```
[edit system scripts op traceoptions file <filename>]
files number size size;
```

For example, set the maximum file size to 640 KB and the maximum number of files to 20. When the file that receives the output of the tracing operation (***filename***) reaches 640 KB, it is renamed and compressed to ***filename*.0.gz**, and a new file called ***filename*** is created. When ***filename*** reaches 640 KB, ***filename*.0.gz** is renamed ***filename*.1.gz** and ***filename*** is renamed and compressed to ***filename*.0.gz**. This process repeats until there are 20 trace files. Then the oldest file (***filename*.19.gz**) is overwritten.

The number of files can range from 2 through 1000 files. The file size can range from 10 KB through 1 gigabyte (GB).

NOTE:

If you set either a maximum file size or a maximum number of trace files, you also must specify the other parameter and a filename.

Configuring Access to Op Script Log Files

By default, access to the op script log file is restricted to the owner. You can manually configure access by including the **world-readable** or **no-world-readable** statement at the **[edit system scripts op traceoptions file <*filename*>]** hierarchy level.

```
[edit system scripts op traceoptions file <filename>]
(world-readable | no-world-readable);
```

The **no-world-readable** statement restricts op script log access to the owner. The **world-readable** statement enables unrestricted access to the op script log file.

Configuring the Op Script Trace Operations

By default, only important events are logged. You can configure the trace operations to be logged by including the following statements at the **[edit system scripts op traceoptions]** hierarchy level:

```
[edit system scripts op traceoptions]
flag all;
flag events;
flag input;
flag offline;
flag output;
flag rpc;
flag xslt;
```

Table 2 on page 88 describes the meaning of the op script tracing flags.

Table 2: Op Script Tracing Flags

Flag	Description	Default Setting
all	Trace all operations.	Off
events	Trace important events.	On
input	Trace op script input data.	Off
offline	Generate data for offline development.	Off
output	Trace op script output data.	Off
rpc	Trace op script RPCs.	Off
xslt	Trace the Extensible Stylesheet Language Transformations (XSLT) library.	Off

PART 5

Index

Index

R

S

T

X