



---

Junos<sup>®</sup> OS

# Configuration and Operations Automation Overview

Release

14.1



---

Published: 2014-05-08

Juniper Networks, Inc.  
1194 North Mathilda Avenue  
Sunnyvale, California 94089  
USA  
408-745-2000  
[www.juniper.net](http://www.juniper.net)

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

Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

*Junos<sup>®</sup> OS Configuration and Operations Automation Overview*

14.1

Copyright © 2014, Juniper Networks, Inc.  
All rights reserved.

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

#### YEAR 2000 NOTICE

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

#### END USER LICENSE AGREEMENT

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

	About the Documentation . . . . .	xv
	Documentation and Release Notes . . . . .	xv
	Supported Platforms . . . . .	xv
	Using the Examples in This Manual . . . . .	xvi
	Merging a Full Example . . . . .	xvi
	Merging a Snippet . . . . .	xvii
	Documentation Conventions . . . . .	xvii
	Documentation Feedback . . . . .	xix
	Requesting Technical Support . . . . .	xx
	Self-Help Online Tools and Resources . . . . .	xx
	Opening a Case with JTAC . . . . .	xx
<b>Part 1</b>	<b>Overview</b>	
<b>Chapter 1</b>	<b>Introduction to Junos Automation . . . . .</b>	<b>3</b>
	Junos Automation Overview . . . . .	3
	Junos Configuration Automation: Commit Scripts . . . . .	4
	Junos Operations Automation: Op Scripts . . . . .	4
	Junos Event Automation: Event Scripts and Event Policy . . . . .	4
	Event Policy . . . . .	4
	Event Scripts . . . . .	5
<b>Chapter 2</b>	<b>Junos XML API and Junos XML Management Protocol . . . . .</b>	<b>7</b>
	Junos XML API and Junos XML Management Protocol Overview . . . . .	7
	Advantages of Using the Junos XML Management Protocol and Junos XML	
	API . . . . .	8
	Parsing Device Output . . . . .	8
	Displaying Device Output . . . . .	9
<b>Chapter 3</b>	<b>Extension Functions, Templates, and Parameters in the jcs and slax</b>	
	<b>Namespaces . . . . .</b>	<b>11</b>
	Junos Script Automation: Understanding Extension Functions in the jcs and slax	
	Namespaces . . . . .	11
	Summary of Extension Functions in the jcs and slax Namespaces . . . . .	13
	Extension Functions in the jcs and slax Namespaces . . . . .	16
	base64-decode() Function (slax Namespace) . . . . .	17
	base64-encode() Function (slax Namespace) . . . . .	17
	break-lines() Function (jcs and slax Namespaces) . . . . .	17
	close() Function (jcs Namespace) . . . . .	18
	dampen() Function (jcs and slax Namespaces) . . . . .	18
	document() Function (slax Namespace) . . . . .	19
	empty() Function (jcs and slax Namespaces) . . . . .	20

	evaluate() Function (slax Namespace) . . . . .	21
	execute() Function (jcs Namespace) . . . . .	21
	first-of() Function (jcs and slax Namespaces) . . . . .	22
	get-command() Function (jcs and slax Namespaces) . . . . .	23
	get-hello() Function (jcs Namespace) . . . . .	24
	get-input() Function (jcs and slax Namespaces) . . . . .	25
	get-protocol() Function (jcs Namespace) . . . . .	25
	get-secret() Function (jcs and slax Namespaces) . . . . .	26
	hostname() Function (jcs Namespace) . . . . .	27
	invoke() Function (jcs Namespace) . . . . .	27
	open() Function (jcs Namespace) . . . . .	28
	output() Function (jcs and slax Namespaces) . . . . .	30
	parse-ip() Function (jcs Namespace) . . . . .	31
	printf() Function (jcs and slax Namespaces) . . . . .	32
	progress() Function (jcs and slax Namespaces) . . . . .	33
	regex() Function (jcs and slax Namespaces) . . . . .	34
	sleep() Function (jcs and slax Namespaces) . . . . .	35
	split() Function (jcs and slax Namespaces) . . . . .	36
	sysctl() Function (jcs and slax Namespaces) . . . . .	37
	syslog() Function (jcs and slax Namespaces) . . . . .	37
	trace() Function (jcs and slax Namespaces) . . . . .	39
	Junos Script Automation: Named Templates in the jcs Namespace Overview . .	40
	Junos Named Templates in the jcs Namespace Summary . . . . .	41
	Junos Named Templates in the jcs Namespace . . . . .	42
	jcs:edit-path Template . . . . .	42
	jcs:emit-change Template . . . . .	43
	jcs:emit-comment Template . . . . .	45
	jcs:grep Template . . . . .	46
	jcs:load-configuration Template . . . . .	47
	jcs:statement Template . . . . .	50
	Junos Script Automation: Global Parameters and Variables in the junos.xsl	
	File . . . . .	51
	Global Parameters . . . . .	51
	Global Variable . . . . .	52
<b>Chapter 4</b>	<b>XML . . . . .</b>	<b>57</b>
	XML Overview . . . . .	57
	Tag Elements . . . . .	57
	Attributes . . . . .	58
	Namespaces . . . . .	58
	Document Type Definition . . . . .	59
	XML and Junos OS . . . . .	59
<b>Chapter 5</b>	<b>XSLT . . . . .</b>	<b>63</b>
	XSLT Overview . . . . .	63
	XSLT Advantages . . . . .	63
	XSLT Engine . . . . .	64
	XSLT Concepts . . . . .	64
	XSLT Namespace . . . . .	65

	XPath Overview . . . . .	65
	Nodes and Axes . . . . .	66
	Path and Predicate Syntax . . . . .	66
	XPath Operators . . . . .	67
	XSLT Templates Overview . . . . .	68
	Unnamed (Match) Templates . . . . .	68
	Named Templates . . . . .	69
	XSLT Parameters Overview . . . . .	70
	Declaring Parameters . . . . .	70
	Passing Parameters . . . . .	71
	Example: Parameters and Match Templates . . . . .	72
	Example: Parameters and Named Templates . . . . .	72
	XSLT Variables Overview . . . . .	73
	XSLT Programming Instructions Overview . . . . .	74
	<xsl:choose> Programming Instruction . . . . .	74
	<xsl:for-each> Programming Instruction . . . . .	75
	<xsl:if> Programming Instruction . . . . .	75
	Sample XSLT Programming Instructions and Pseudocode . . . . .	76
	XSLT Recursion Overview . . . . .	77
	XSLT Context (Dot) Overview . . . . .	78
<b>Chapter 6</b>	<b>SLAX . . . . .</b>	<b>79</b>
	SLAX Overview . . . . .	79
	SLAX Advantages . . . . .	79
	How SLAX Works . . . . .	80
	Converting Scripts Between SLAX and XSLT . . . . .	81
	Converting a Script from SLAX to XSLT . . . . .	81
	Converting a Script from XSLT to SLAX . . . . .	82
	SLAX Syntax Rules Overview . . . . .	83
	Code Blocks . . . . .	83
	Comments . . . . .	84
	Line Termination . . . . .	84
	Strings . . . . .	85
	SLAX Elements and Element Attributes Overview . . . . .	85
	SLAX Elements . . . . .	85
	SLAX Element Attributes . . . . .	86
	XPath Expressions Overview for SLAX . . . . .	87
	SLAX Templates Overview . . . . .	88
	Unnamed (Match) Templates . . . . .	88
	Named Templates . . . . .	89
	SLAX Functions Overview . . . . .	91
	SLAX Parameters Overview . . . . .	93
	Declaring Parameters . . . . .	93
	Passing Parameters to Templates . . . . .	94
	Example: Parameters and Match Templates . . . . .	95
	Passing Parameters to Functions . . . . .	96
	SLAX Variables Overview . . . . .	97
	Immutable variables . . . . .	97
	Mutable variables . . . . .	98

	SLAX Statements Overview . . . . .	100
	for-each Statement . . . . .	101
	if, else if, and else Statements . . . . .	101
	match Statement . . . . .	102
	ns Statement . . . . .	103
	version Statement . . . . .	103
	XSLT Elements Without SLAX Equivalents . . . . .	104
	SLAX Operators . . . . .	104
<b>Chapter 7</b>	<b>libslax . . . . .</b>	<b>109</b>
	libslax Distribution Overview . . . . .	109
	libslax Library and Extension Libraries Overview . . . . .	110
	libslax Library . . . . .	110
	libslax Extension Libraries . . . . .	110
	Understanding the SLAX Processor (slaxproc) . . . . .	111
	slaxproc Overview . . . . .	111
	slaxproc Modes . . . . .	112
	slaxproc Options . . . . .	112
	slaxproc File Argument Handling . . . . .	114
	slaxproc UNIX Scripting Support . . . . .	114
<b>Part 2</b>	<b>Configuration</b>	
<b>Chapter 8</b>	<b>Storing and Enabling Scripts . . . . .</b>	<b>119</b>
	Storing and Enabling Scripts . . . . .	119
	Storing Scripts in Flash Memory . . . . .	120
	Storing and Using Imported Scripts and Script Functionality . . . . .	121
<b>Chapter 9</b>	<b>Configuring a Remote Source for Scripts . . . . .</b>	<b>123</b>
	Overview of Updating Scripts from a Remote Source . . . . .	123
	Using a Master Source Location for a Script . . . . .	124
	Configuring and Refreshing from the Master Source for a Script . . . . .	125
	Configuring the Master Source for a Script . . . . .	125
	Updating a Script from the Master Source . . . . .	125
	Example: Configuring and Refreshing from the Master Source for a Script . . . . .	127
	Using an Alternate Source Location for a Script . . . . .	129
	Refreshing a Script from an Alternate Location . . . . .	129
	Example: Refreshing a Script from an Alternate Source . . . . .	131
<b>Chapter 10</b>	<b>Configuring Script Synchronization Between Routing Engines . . . . .</b>	<b>135</b>
	Example: Configuring Script Synchronization Between Routing Engines . . . . .	135
<b>Chapter 11</b>	<b>Configuring the Session Protocol for Scripts . . . . .</b>	<b>139</b>
	Specifying the Session Protocol for Connections Using Junos Automation Scripts . . . . .	139
	Session Protocol in Junos Automation Scripts Overview . . . . .	139
	Example: Specifying the Session Protocol for a Connection Using an Automation Script . . . . .	141

<b>Chapter 12</b>	<b>Configuring Limits on Event Policies and Memory Allocation for Scripts</b>	<b>153</b>
	Example: Configuring Limits on Executed Event Policies and Memory Allocation for Scripts	153
	Overview of Limits on Executed Event Policies and Memory Allocation for Scripts	153
	Example: Configuring Limits on Executed Event Policies and Memory Allocation for Scripts	154
<b>Chapter 13</b>	<b>Provisioning Services Using Service Template Automation</b>	<b>157</b>
	Example: Service Template Automation	157
	Service Template Automation Overview	157
	Example: Configuring Service Template Automation	158
<b>Chapter 14</b>	<b>Using libslax</b>	<b>171</b>
	Downloading and Installing the libslax Distribution	171
	libslax Default Extension Libraries: bit, cURL, and xutil	171
	libslax bit Extension Library	172
	libslax cURL Extension Library	174
	Understanding the cURL Extension Library	174
	curl:close	177
	curl:open	177
	curl:perform	177
	curl:set	178
	curl:single	179
	cURL Examples	179
	libslax xutil Extension Library	181
	SLAX Debugger, Profiler, and callflow	181
	SLAX Debugger, Profiler, and callflow Overview	181
	Using the SLAX Debugger, Profiler, and callflow	183
	Invoking the SLAX Debugger	183
	Using the SLAX Debugger (sdb)	184
	Using the SLAX Profiler	185
	Using callflow	187
	Using the SLAX Processor (slaxproc)	187
	Validating SLAX Script Syntax	188
	Converting Scripts Between XSLT and SLAX Formats	188
	Running SLAX Scripts	190
	Formatting SLAX Scripts	191
<b>Chapter 15</b>	<b>SLAX Statements</b>	<b>193</b>
	append	194
	apply-imports	195
	apply-templates	195
	attribute	196
	attribute-set	197
	call	199
	copy-node	200
	copy-of	201
	decimal-format	201

	element . . . . .	203
	else . . . . .	203
	else if . . . . .	204
	expr . . . . .	205
	fallback . . . . .	206
	for . . . . .	207
	for-each . . . . .	208
	function . . . . .	210
	if . . . . .	211
	import . . . . .	212
	key . . . . .	213
	match . . . . .	215
	message . . . . .	216
	mode . . . . .	216
	mvar . . . . .	217
	number . . . . .	218
	output-method . . . . .	222
	param . . . . .	225
	preserve-space . . . . .	226
	priority . . . . .	226
	processing-instruction . . . . .	227
	result . . . . .	229
	set . . . . .	230
	sort . . . . .	230
	strip-space . . . . .	232
	template . . . . .	232
	terminate . . . . .	234
	trace . . . . .	234
	uexpr . . . . .	235
	use-attribute-sets . . . . .	236
	var . . . . .	237
	version . . . . .	237
	while . . . . .	238
	with . . . . .	239
<b>Chapter 16</b>	<b>Standard XPath and XSLT Functions Used in Automation Scripts . . . . .</b>	<b>241</b>
	concat() . . . . .	241
	contains() . . . . .	242
	count() . . . . .	242
	last() . . . . .	242
	name() . . . . .	243
	not() . . . . .	243
	position() . . . . .	243
	starts-with() . . . . .	244
	string-length() . . . . .	244
	substring-after() . . . . .	245
	substring-before() . . . . .	245



<b>Chapter 17</b>	<b>Standard XSLT Elements and Attributes Used in Automation Scripts . . .</b>	<b>247</b>
	xsl:apply-templates . . . . .	248
	xsl:call-template . . . . .	248
	xsl:choose . . . . .	249
	xsl:comment . . . . .	250
	xsl:copy-of . . . . .	250
	xsl:element . . . . .	251
	xsl:for-each . . . . .	251
	xsl:if . . . . .	252
	xsl:import . . . . .	252
	xsl:otherwise . . . . .	253
	xsl:param . . . . .	254
	xsl:stylesheet . . . . .	255
	xsl:template . . . . .	256
	xsl:template match="/" Template . . . . .	257
	xsl:text . . . . .	259
	xsl:value-of . . . . .	259
	xsl:variable . . . . .	260
	xsl:when . . . . .	261
	xsl:with-param . . . . .	261
<b>Chapter 18</b>	<b>Configuration Statements Common to All Scripts . . . . .</b>	<b>263</b>
	load-scripts-from-flash (Scripts) . . . . .	263
	max-datasize . . . . .	264
	synchronize (Scripts) . . . . .	265
<b>Part 3</b>	<b>Administration</b>	
<b>Chapter 19</b>	<b>Synchronizing Scripts Between Routing Engines . . . . .</b>	<b>269</b>
	Synchronizing Scripts Between Routing Engines . . . . .	269
	Understanding Script Synchronization Between Routing Engines . . . . .	269
	Script Synchronization Between Routing Engines . . . . .	270
	Configuring Script Synchronization Between Routing Engines for Commit	
	Synchronize Operations . . . . .	271
	Synchronizing Scripts Between Routing Engines on a Per-Commit	
	Basis . . . . .	271
	Synchronizing Scripts Between Routing Engines from Operational	
	Mode . . . . .	272
	Synchronizing a Script Between Routing Engines After a Refresh . . . . .	273
<b>Chapter 20</b>	<b>Operational Commands . . . . .</b>	<b>275</b>
	op invoke-debugger cli . . . . .	276
	request system scripts convert . . . . .	277
	request system scripts refresh-from . . . . .	279
	request system scripts synchronize . . . . .	281
<b>Part 4</b>	<b>Index</b>	
	Index . . . . .	285



# List of Figures

<b>Part 1</b>	<b>Overview</b>	
<b>Chapter 5</b>	<b>XSLT</b> .....	<b>63</b>
	Figure 1: Flow of XSLT Commit Script Through the XSLT Engine .....	64
<b>Chapter 6</b>	<b>SLAX</b> .....	<b>79</b>
	Figure 2: SLAX Script Input and Output .....	80
<b>Part 2</b>	<b>Configuration</b>	
<b>Chapter 17</b>	<b>Standard XSLT Elements and Attributes Used in Automation Scripts</b> . . .	<b>247</b>
	Figure 3: Commit Script Input and Output .....	257



# List of Tables

	<b>About the Documentation</b> . . . . .	<b>xv</b>
	Table 1: Notice Icons . . . . .	xviii
	Table 2: Text and Syntax Conventions . . . . .	xviii
<b>Part 1</b>	<b>Overview</b>	
<b>Chapter 3</b>	<b>Extension Functions, Templates, and Parameters in the jcs and slax Namespaces</b> . . . . .	<b>11</b>
	Table 3: Extension Functions in the jcs and slax Namespaces . . . . .	13
	Table 4: Options for slax:document Function . . . . .	20
	Table 5: Facility Strings . . . . .	38
	Table 6: Severity Strings . . . . .	39
	Table 7: Junos Named Templates . . . . .	41
	Table 8: Predefined Parameters Available to Automation Scripts . . . . .	51
	Table 9: Global Variable \$junos-context Available to Automation Scripts . . . . .	54
<b>Chapter 5</b>	<b>XSLT</b> . . . . .	<b>63</b>
	Table 10: XSLT Concepts . . . . .	64
	Table 11: Examples and Pseudocode for XSLT Variable Declaration . . . . .	74
	Table 12: Examples and Pseudocode for XSLT Programming Instructions . . . . .	76
<b>Chapter 6</b>	<b>SLAX</b> . . . . .	<b>79</b>
	Table 13: SLAX Operators . . . . .	105
<b>Chapter 7</b>	<b>libslax</b> . . . . .	<b>109</b>
	Table 14: slaxproc Modes . . . . .	112
	Table 15: slaxproc Common Options and File Options . . . . .	112
<b>Part 2</b>	<b>Configuration</b>	
<b>Chapter 14</b>	<b>Using libslax</b> . . . . .	<b>171</b>
	Table 16: Functions in the bit Extension Library . . . . .	172
	Table 17: Functions in the cURL Extension Library . . . . .	174
	Table 18: Web Services Elements in the cURL Extension Library . . . . .	175
	Table 19: E-mail Elements in the cURL Extension Library . . . . .	176
	Table 20: curl:perform Reply Elements . . . . .	178
	Table 21: curl:perform <header> Elements . . . . .	178
	Table 22: Functions in the xutil Extension Library . . . . .	181
	Table 23: SLAX Debugger Commands . . . . .	182
	Table 24: Profile command options . . . . .	185
<b>Chapter 15</b>	<b>SLAX Statements</b> . . . . .	<b>193</b>

Table 25: Numbering Styles for SLAX Statement number, format Option . . . . . 219

# About the Documentation

- [Documentation and Release Notes on page xv](#)
- [Supported Platforms on page xv](#)
- [Using the Examples in This Manual on page xvi](#)
- [Documentation Conventions on page xvii](#)
- [Documentation Feedback on page xix](#)
- [Requesting Technical Support on page xx](#)

## Documentation and Release Notes

---

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

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

Juniper Networks Books publishes books by Juniper Networks engineers and subject matter experts. These books go beyond the technical documentation to explore the nuances of network architecture, deployment, and administration. The current list can be viewed at <http://www.juniper.net/books>.

## Supported Platforms

---

For the features described in this document, the following platforms are supported:

- [ACX Series](#)
- [EX Series](#)
- [J Series](#)
- [M Series](#)
- [MX Series](#)
- [QFabric System](#)
- [QFX Series standalone switches](#)
- [SRX Series](#)

- [T Series](#)
- [PTX Series](#)

## Using the Examples in This Manual

---

If you want to use the examples in this manual, you can use the **load merge** or the **load merge relative** command. These commands cause the software to merge the incoming configuration into the current candidate configuration. The example does not become active until you commit the candidate configuration.

If the example configuration contains the top level of the hierarchy (or multiple hierarchies), the example is a *full example*. In this case, use the **load merge** command.

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

### Merging a Full Example

To merge a full example, follow these steps:

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

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

```
system {
  scripts {
    commit {
      file ex-script.xml;
    }
  }
}
interfaces {
  fxp0 {
    disable;
    unit 0 {
      family inet {
        address 10.0.0.1/24;
      }
    }
  }
}
```

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

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



## Merging a Snippet

To merge a snippet, follow these steps:

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

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

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

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

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

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

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

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

---

## Documentation Conventions

[Table 1 on page xviii](#) defines notice icons used in this guide.

Table 1: Notice Icons

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

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

Table 2: Text and Syntax Conventions

Convention	Description	Examples
<b>Bold text like this</b>	Represents text that you type.	To enter configuration mode, type the <b>configure</b> command:  user@host> <b>configure</b>
Fixed-width text like this	Represents output that appears on the terminal screen.	user@host> <b>show chassis alarms</b>  No alarms currently active
<i>Italic text like this</i>	<ul style="list-style-type: none"> <li>Introduces or emphasizes important new terms.</li> <li>Identifies guide names.</li> <li>Identifies RFC and Internet draft titles.</li> </ul>	<ul style="list-style-type: none"> <li>A policy <i>term</i> is a named structure that defines match conditions and actions.</li> <li><i>Junos OS CLI User Guide</i></li> <li>RFC 1997, <i>BGP Communities Attribute</i></li> </ul>
<i>Italic text like this</i>	Represents variables (options for which you substitute a value) in commands or configuration statements.	Configure the machine's domain name:  [edit] root@# <b>set system domain-name</b> <i>domain-name</i>

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

Convention	Description	Examples
Text like this	Represents names of configuration statements, commands, files, and directories; configuration hierarchy levels; or labels on routing platform components.	<ul style="list-style-type: none"><li>To configure a stub area, include the <b>stub</b> statement at the <b>[edit protocols ospf area area-id]</b> hierarchy level.</li><li>The console port is labeled <b>CONSOLE</b>.</li></ul>
< > (angle brackets)	Encloses optional keywords or variables.	<b>stub &lt;default-metric <i>metric</i>&gt;;</b>
(pipe symbol)	Indicates a choice between the mutually exclusive keywords or variables on either side of the symbol. The set of choices is often enclosed in parentheses for clarity.	<b>broadcast   multicast</b>  <b>(<i>string1</i>   <i>string2</i>   <i>string3</i>)</b>
# (pound sign)	Indicates a comment specified on the same line as the configuration statement to which it applies.	<b>rsvp { # Required for dynamic MPLS only</b>
[ ] (square brackets)	Encloses a variable for which you can substitute one or more values.	<b>community name members [ <i>community-ids</i> ]</b>
Indentation and braces ( { } )	Identifies a level in the configuration hierarchy.	<pre>[edit] routing-options {   static {     route default {       nexthop <i>address</i>;       retain;     }   } }</pre>
;(semicolon)	Identifies a leaf statement at a configuration hierarchy level.	
GUI Conventions		
Bold text like this	Represents graphical user interface (GUI) items you click or select.	<ul style="list-style-type: none"><li>In the Logical Interfaces box, select <b>All Interfaces</b>.</li><li>To cancel the configuration, click <b>Cancel</b>.</li></ul>
> (bold right angle bracket)	Separates levels in a hierarchy of menu selections.	In the configuration editor hierarchy, select <b>Protocols&gt;Ospf</b> .

## Documentation Feedback

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

- Document or topic name
- URL or page number

- Software release version (if applicable)

## Requesting Technical Support

---

Technical product support is available through the Juniper Networks Technical Assistance Center (JTAC). If you are a customer with an active J-Care or JNASC support contract, or are covered under warranty, and need post-sales technical support, you can access our tools and resources online or open a case with JTAC.

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

## Self-Help Online Tools and Resources

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

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

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

## Opening a Case with JTAC

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

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

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



## PART 1

# Overview

- [Introduction to Junos Automation on page 3](#)
- [Junos XML API and Junos XML Management Protocol on page 7](#)
- [Extension Functions, Templates, and Parameters in the jcs and slax Namespaces on page 11](#)
- [XML on page 57](#)
- [XSLT on page 63](#)
- [SLAX on page 79](#)
- [libslax on page 109](#)





## CHAPTER 1

# Introduction to Junos Automation

- [Junos Automation Overview on page 3](#)

## Junos Automation Overview

---

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

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

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

Junos automation scripts can be written in either of two scripting languages: Extensible Stylesheet Language Transformations (XSLT) or Stylesheet Language Alternative Syntax (SLAX). XSLT is a standard for processing Extensible Markup Language (XML) data and is designed to convert one XML document into another. SLAX is an alternative to XSLT. It has a simple syntax that follows the style of C and PERL, but retains the same semantics as XSLT. Programmers who are familiar with C often find it easier to learn and use SLAX. Scripts written in one language are easily converted to the other.

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

- [Junos Configuration Automation: Commit Scripts on page 4](#)
- [Junos Operations Automation: Op Scripts on page 4](#)
- [Junos Event Automation: Event Scripts and Event Policy on page 4](#)

## Junos Configuration Automation: Commit Scripts

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

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

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

## Junos Operations Automation: Op Scripts

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

- Create custom operational mode commands
- Execute a series of operational mode commands
- Customize the output of operational mode commands
- Shorten troubleshooting time by gathering operational information and iteratively narrowing down the cause of a network problem
- Perform controlled configuration changes
- Monitor the overall status of a device by creating a general operation script that periodically checks network warning parameters, such as high CPU usage.

## Junos Event Automation: Event Scripts and Event Policy

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

### Event Policy

---

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

- Ignore the event
- Upload a file to a specified destination

- Execute Junos OS operational mode commands
- Execute Junos OS event scripts

### Event Scripts

---

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

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

#### Related Documentation

- *Commit Script Overview*



## CHAPTER 2

# Junos XML API and Junos XML Management Protocol

- [Junos XML API and Junos XML Management Protocol Overview on page 7](#)
- [Advantages of Using the Junos XML Management Protocol and Junos XML API on page 8](#)

## Junos XML API and Junos XML Management Protocol Overview

---

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

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

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

You can use the Junos XML management protocol and Junos XML API to configure devices running Junos OS or request information about the device configuration or operation.

You can write client applications to interact with the Junos XML protocol server, and you can also utilize the Junos XML protocol to build custom end-user interfaces for configuration and information retrieval and display, such as a Web browser-based interface.

- Related Documentation**
- [Advantages of Using the Junos XML Management Protocol and Junos XML API on page 8](#)
  - [XML and Junos OS on page 59](#)
  - [XML Overview on page 57](#)

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

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

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

### Parsing Device Output

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

The formatted ASCII follows:

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

The corresponding XML-tagged version is:

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

When a client application needs to extract a specific value from formatted ASCII output, it must rely on the value's location, expressed either absolutely or with respect to labels or values in adjacent fields. Suppose that the client application wants to extract the interface index. It can use a regular-expression matching utility to locate specific strings, but one difficulty is that the number of digits in the interface index is not necessarily predictable. The client application cannot simply read a certain number of characters

after the **Interface index:** label, but must instead extract everything between the label and the subsequent label **SNMP ifIndex:** and also account for the included comma.

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

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

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

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

## Displaying Device Output

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

### Related Documentation

- [Junos XML API and Junos XML Management Protocol Overview on page 7](#)
- [XML Overview on page 57](#)





## CHAPTER 3

# Extension Functions, Templates, and Parameters in the jcs and slax Namespaces

- [Junos Script Automation: Understanding Extension Functions in the jcs and slax Namespaces on page 11](#)
- [Summary of Extension Functions in the jcs and slax Namespaces on page 13](#)
- [Extension Functions in the jcs and slax Namespaces on page 16](#)
- [Junos Script Automation: Named Templates in the jcs Namespace Overview on page 40](#)
- [Junos Named Templates in the jcs Namespace Summary on page 41](#)
- [Junos Named Templates in the jcs Namespace on page 42](#)
- [Junos Script Automation: Global Parameters and Variables in the junos.xml File on page 51](#)

## Junos Script Automation: Understanding Extension Functions in the jcs and slax Namespaces

---

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

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

The following SLAX namespace statement maps the **jcs** prefix to the namespace URI that defines Junos OS extension functions used in commit, op, and event scripts:

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

SLAX extension functions are defined in the namespace with the associated URI `http://xml.libslax.org/slax`. To use SLAX extension functions in scripts, you must include the namespace URI in your style sheet declaration. Generally, the **slax** prefix is mapped to the URI, and you then use the extension functions by prepending the **slax** prefix to the function name. During processing, the **slax** prefix is expanded into its associated URI reference.

The following SLAX namespace statement maps the **slax** prefix to the namespace URI that defines SLAX extension functions:

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

The **slax** namespace is supported starting with Junos OS Release 12.2. Scripts using Junos OS-independent extension functions that existed in earlier releases in the **jcs** namespace can use either the **jcs** or the **slax** namespace starting with Junos OS Release 12.2. However, to use any of these functions in earlier Junos OS releases, scripts must use the **jcs** namespace URI. For a list of available functions and associated namespaces, see [“Summary of Extension Functions in the jcs and slax Namespaces” on page 13](#).

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

The following example maps the **jcs** prefix to the namespace identified by the URI `http://xml.juniper.net/junos/commit-scripts/1.0`. The script then calls the **jcs:invoke()** function with one argument.

<b>XSLT Syntax</b>	<pre>&lt;?xml version="1.0"?&gt; &lt;xsl:stylesheet version="1.0"   xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0"&gt;   ...   &lt;xsl:variable name="result" select="jcs:invoke(\$command)"/&gt;   ... &lt;/xsl:stylesheet&gt;</pre>
--------------------	--

<b>SLAX Syntax</b>	<pre>version 1.0; ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0"; ... var \$result = jcs:invoke(\$command); ...</pre>
--------------------	---

The following example maps the **slax** prefix to the namespace identified by the URI `http://xml.libslax.org/slax`. The script then calls the **slax:get-input()** function with one string argument. The version statement specifies version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.

<b>SLAX Syntax</b>	<pre>version 1.1; ns slax = "http://xml.libslax.org/slax"; ...</pre>
--------------------	--

```
var $input = slax:get-input($prompt);
...
```

#### Related Documentation

- [Summary of Extension Functions in the jcs and slax Namespaces on page 13](#)
- [Junos Script Automation: Named Templates in the jcs Namespace Overview on page 40](#)
- [Junos Script Automation: Global Parameters and Variables in the junos.xml File on page 51](#)
- [SLAX Variables Overview on page 97](#)
- [XSLT Variables Overview on page 73](#)

## Summary of Extension Functions in the jcs and slax Namespaces

The Junos OS and SLAX extension functions are summarized in [Table 3 on page 13](#).

A function in the **jcs** namespace is defined in the namespace URI <http://xml.juniper.net/junos/commit-scripts/1.0>.

A function in the **slax** namespace is defined in the namespace URI <http://xml.libslax.org/slax>. Functions using the **slax** namespace are supported starting with Junos OS Release 12.2.

Functions introduced in version 1.0 of the SLAX language can be used in SLAX scripts that include either a "version 1.0" statement or a "version 1.1" statement, but functions introduced in version 1.1 of the SLAX language can only be used in SLAX scripts that include the "version 1.1" statement. Scripts written in version 1.1 of the SLAX language are supported starting in Junos OS Release 12.2.

**Table 3: Extension Functions in the jcs and slax Namespaces**

Function	Name-spaces	SLAX Version	Type	Description
<a href="#">base64-decode()</a>	slax	1.1	Data manipulation	Decode BASE64 encoded data and return a string.
<a href="#">base64-encode()</a>	slax	1.1	Data manipulation	Encode a string of data in the BASE64 encoding format.
<a href="#">break-lines()</a>	jcs, slax	1.0	Data manipulation	Break a simple element into multiple elements, delimited by newlines.
<a href="#">close()</a>	jcs	1.0	Utility	Close a previously opened connection handle.
<a href="#">dampen()</a>	jcs, slax	1.0	Utility	Prevent the same operation from being repeatedly executed within a script.

Table 3: Extension Functions in the jcs and slax Namespaces (*continued*)

Function	Name-spaces	SLAX Version	Type	Description
<code>document()</code>	slax	1.1	Input/output control	Read data from a file or URL and return a string.
<code>empty()</code>	jcs, slax	1.0	Logic	Evaluate a node set or string argument to determine if it is an empty value.
<code>evaluate()</code>	slax	1.1	Input/output control	Evaluate a SLAX expression and return the result.
<code>execute()</code>	jcs	1.0	Utility	Execute a remote procedure call (RPC) within the context of a specified connection handle.
<code>first-of()</code>	jcs, slax	1.0	Logic	Return the first nonempty (non-null) item in a list. If all objects in the list are empty, the default expression is returned.
<code>get-command()</code>	jcs, slax	1.1	Input/output control	Prompt the user for command input and return the input as a string.
<code>get-hello()</code>	jcs	1.0	Utility	Return the session ID and the capabilities of the NETCONF server during a NETCONF session.
<code>get-input()</code>	jcs, slax	1.0	Input/output control	Invoke a CLI prompt and wait for user input. If the script is run non-interactively, the function returns an empty value. This function cannot be used with event scripts.
<code>get-protocol()</code>	jcs	1.0	Utility	Return the session protocol associated with the connection handle.
<code>get-secret()</code>	jcs, slax	1.0	Input/output control	Invoke a CLI prompt and wait for user input. The input is not echoed back to the user.
<code>hostname()</code>	jcs	1.0	Utility	Return the fully qualified domain name associated with a given IPv4 or IPv6 address, provided the DNS server is configured on the device.
<code>invoke()</code>	jcs	1.0	Utility	Invoke an RPC on a local device running Junos OS.

Table 3: Extension Functions in the jcs and slax Namespaces (*continued*)

Function	Name-spaces	SLAX Version	Type	Description
<code>open()</code>	jcs	1.0	Utility	Return a connection handle that can be used to execute RPCs.
<code>output()</code>	jcs, slax	1.0	Input/output control	Generate unformatted output text that is immediately sent to the CLI session.
<code>parse-ip()</code>	jcs	1.0	Data manipulation	Parse an IPv4 or IPv6 address and return the host IP address, protocol family, prefix length, network address, and network mask.
<code>printf()</code>	jcs, slax	1.0	Input/output control	Generate formatted output text. Most standard <code>printf</code> formats are supported, in addition to some Junos OS-specific formats. The function returns a formatted string but does not print it on call.
<code>progress()</code>	jcs, slax	1.0	Input/output control	Issue a progress message containing the single argument immediately to the CLI session provided that the <b>detail</b> flag was specified when the script was invoked.
<code>regex()</code>	jcs, slax	1.0	Data manipulation	Evaluate a regular expression against a given string argument and return any matches.
<code>sleep()</code>	jcs, slax	1.0	Utility	Cause the script to sleep for a specified time.
<code>split()</code>	jcs, slax	1.0	Data manipulation	Split a string into an array of substrings delimited by a regular expression pattern.
<code>sysctl()</code>	jcs, slax	1.0	Utility	Return the value of the given <b>sysctl</b> value as a string or an integer.
<code>syslog()</code>	jcs, slax	1.0	Input/output control	Log messages with the specified priority to the system log file.
<code>trace()</code>	jcs, slax	1.0	Input/output control	Issue a trace message, which is sent to the trace file.

#### Related Documentation

- [Junos Named Templates in the jcs Namespace Summary on page 41](#)

- [Junos Script Automation: Understanding Extension Functions in the jcs and slax Namespaces on page 11](#)
- [Junos Script Automation: Named Templates in the jcs Namespace Overview on page 40](#)
- [Junos Script Automation: Global Parameters and Variables in the junos.xml File on page 51](#)

---

## Extension Functions in the jcs and slax Namespaces

---

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

- [base64-decode\(\) Function \(slax Namespace\) on page 17](#)
- [base64-encode\(\) Function \(slax Namespace\) on page 17](#)
- [break-lines\(\) Function \(jcs and slax Namespaces\) on page 17](#)
- [close\(\) Function \(jcs Namespace\) on page 18](#)
- [dampen\(\) Function \(jcs and slax Namespaces\) on page 18](#)
- [document\(\) Function \(slax Namespace\) on page 19](#)
- [empty\(\) Function \(jcs and slax Namespaces\) on page 20](#)
- [evaluate\(\) Function \(slax Namespace\) on page 21](#)
- [execute\(\) Function \(jcs Namespace\) on page 21](#)
- [first-of\(\) Function \(jcs and slax Namespaces\) on page 22](#)
- [get-command\(\) Function \(jcs and slax Namespaces\) on page 23](#)
- [get-hello\(\) Function \(jcs Namespace\) on page 24](#)
- [get-input\(\) Function \(jcs and slax Namespaces\) on page 25](#)
- [get-protocol\(\) Function \(jcs Namespace\) on page 25](#)
- [get-secret\(\) Function \(jcs and slax Namespaces\) on page 26](#)
- [hostname\(\) Function \(jcs Namespace\) on page 27](#)
- [invoke\(\) Function \(jcs Namespace\) on page 27](#)
- [open\(\) Function \(jcs Namespace\) on page 28](#)
- [output\(\) Function \(jcs and slax Namespaces\) on page 30](#)
- [parse-ip\(\) Function \(jcs Namespace\) on page 31](#)
- [printf\(\) Function \(jcs and slax Namespaces\) on page 32](#)
- [progress\(\) Function \(jcs and slax Namespaces\) on page 33](#)
- [regex\(\) Function \(jcs and slax Namespaces\) on page 34](#)
- [sleep\(\) Function \(jcs and slax Namespaces\) on page 35](#)
- [split\(\) Function \(jcs and slax Namespaces\) on page 36](#)
- [sysctl\(\) Function \(jcs and slax Namespaces\) on page 37](#)
- [syslog\(\) Function \(jcs and slax Namespaces\) on page 37](#)
- [trace\(\) Function \(jcs and slax Namespaces\) on page 39](#)

### base64-decode() Function (slax Namespace)

<b>Namespaces</b>	<code>http://xml.libslax.org/slax</code>
<b>SLAX Syntax</b>	<code>string slax:base64-decode(string, &lt;control-string&gt;)</code>
<b>Release Information</b>	Function introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	<p>Decode BASE64 encoded data. BASE64 is a means of encoding arbitrary data into a radix-64 format that is more easily transmitted, typically using STMP or HTTP.</p> <p>Include the optional control string argument to replace any non-XML control characters in the decoded string with the specified string. If the argument is an empty string, non-XML characters are removed. The decoded data is returned to the caller.</p>
<b>Parameters</b>	<p><i>control-string</i>—(Optional) String to replace non-XML control characters in the decoded string. Use an empty string argument to remove the non-XML characters.</p> <p><i>string</i>—BASE64 encoded data.</p>
<b>Return Value</b>	<p><i>string</i>—Decoded data.</p> <p>—</p>
<b>Usage Examples</b>	<pre>var \$real-data = slax:base64-decode(\$encoded-data, "@");</pre>

### base64-encode() Function (slax Namespace)

<b>Namespaces</b>	<code>http://xml.libslax.org/slax</code>
<b>SLAX Syntax</b>	<code>string slax:base64-encode(string)</code>
<b>Release Information</b>	Function introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	Encode a string of data in the BASE64 encoding format. BASE64 is a means of encoding arbitrary data into a radix-64 format that is more easily transmitted, typically using STMP or HTTP.
<b>Parameters</b>	<i>string</i> —Input data string.
<b>Return Value</b>	<i>string</i> —Encoded data.
<b>Usage Examples</b>	<pre>var \$encoded-data = slax:base64-encode(\$real-data);</pre>

### break-lines() Function (jcs and slax Namespaces)

<b>Namespaces</b>	<code>http://xml.juniper.net/junos/commit-scripts/1.0</code> <code>http://xml.libslax.org/slax</code>
<b>SLAX Syntax</b>	<code>var \$lines = prefix:break-lines(expression);</code>

<b>XSLT Syntax</b>	<code>&lt;xsl:variable name="lines" select="prefix:break-lines(expression)"/&gt;</code>
<b>Release Information</b>	Function introduced in Junos OS Release 7.6 Support for the slax namespace <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a> added in Junos OS Release 12.2.
<b>Description</b>	Break a simple element into multiple elements, delimited by newlines. This is especially useful for large output elements such as those returned by the <b>show pfe</b> command.  The <i>prefix</i> associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.
<b>Parameters</b>	<i>expression</i> —Original output.
<b>Return Value</b>	<code>\$lines</code> —Output broken up into lines.
<b>Usage Examples</b>	<pre>var \$lines = jcs:break-lines(\$output); for-each (\$lines) {   ... }</pre>

### close() Function (jcs Namespace)

<b>Namespaces</b>	<a href="http://xml.juniper.net/junos/commit-scripts/1.0">http://xml.juniper.net/junos/commit-scripts/1.0</a>
<b>SLAX Syntax</b>	<code>var \$results = jcs:close(connection);</code>
<b>XSLT Syntax</b>	<code>&lt;xsl:variable name="results" select="jcs:close(connection)"/&gt;</code>
<b>Release Information</b>	Function introduced in Junos OS Release 9.3.
<b>Description</b>	Close a previously opened connection handle.
<b>Parameters</b>	<i>connection</i> —Connection handle generated by a call to the <b>jcs:open()</b> function.
<b>Usage Examples</b>	The following example closes the connection handle <code>\$connection</code> , which was originally generated by a call to the <b>jcs:open()</b> function:  <pre>var \$connection = jcs:open(); ... var \$result = jcs:close(\$connection);</pre>

### dampen() Function (jcs and slax Namespaces)

<b>Namespaces</b>	<a href="http://xml.juniper.net/junos/commit-scripts/1.0">http://xml.juniper.net/junos/commit-scripts/1.0</a> <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a>
<b>SLAX Syntax</b>	<code>var \$result = prefix:dampen(tag-string, max, interval);</code>
<b>XSLT Syntax</b>	<code>&lt;xsl:variable name="result" select="prefix:dampen(tag-string, max, interval)"/&gt;</code>
<b>Release Information</b>	Function introduced in Junos OS Release 9.4.



Support for the slax namespace <http://xml.libslax.org/slax> added in Junos OS Release 12.2.

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

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

**Parameters** *interval*—Time interval, in minutes.

*max*—Maximum number of calls to the **jcs:dampen()** function with a given tag allowed before the function returns **false**. This limit is based on the number of calls within a specified time interval.

*tag-string*—Arbitrary string used to distinguish different calls to the **jcs:dampen()** function.

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

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

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

## document() Function (slax Namespace)

**Namespaces** <http://xml.libslax.org/slax>

**SLAX Syntax** *string* slax:document(*url*, <options>)

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

**Description** Read data from a file or URL. The data can be encoded in any character set and can be BASE64 encoded. The default character set is "utf-8". Optional arguments specify the character encoding scheme and the encoding format, and define the replacement string for non-XML control characters. [Table 4 on page 20](#) lists the available options.

**Parameters** options—(Optional) Specify the character encoding scheme and format of the data, and define the replacement string for non-XML control characters.

*url*—File or URL from which to read data.

**Table 4: Options for slax:document Function**

Option	Description
<encoding> <i>string</i>	Character encoding scheme. For example "ascii" or "utf-8".
<format> <i>string</i>	"base64" for BASE64-encoded data.
<non-xml> <i>string</i>	String used to replace non-XML control characters. If the value is an empty string, non-XML characters are removed.

**Return Value** *string*—String representing the data.

**Usage Examples**

```
var $data = slax:document($url);

var $options := {
  <encoding> "ascii";
  <format> "base64";
  <non-xml> "#";
}
var $data2 = slax:document($url, $options);
```

## empty() Function (jcs and slax Namespaces)

**Namespaces** <http://xml.juniper.net/junos/commit-scripts/1.0>  
<http://xml.libslax.org/slax>

**SLAX Syntax** `var $result = prefix:empty(node-set | string);`

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

**Release Information** Function introduced in Junos OS Release 7.6  
 Support for the slax namespace <http://xml.libslax.org/slax> added in Junos OS Release 12.2.

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

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

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

**Return Value** *result*—Boolean value, which is **true** if the argument is empty.

**Usage Examples** In the following example, if `$set` is empty, the script executes the enclosed code block:

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

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

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

### evaluate() Function (slax Namespace)

**Namespaces** `http://xml.libslax.org/slax`

**SLAX Syntax** `object slax:evaluate(expression);`

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

**Description** Evaluate a SLAX expression and return the results of the expression. This permits expressions using the extended syntax provided by SLAX in addition to what is allowed in XPath.

**Parameters** *expression*—SLAX expression to evaluate.

**Return Value** *object*—Result of the expression.

**Usage Examples** `var $result = slax:evaluate("expr[name == '&']");`

### execute() Function (jcs Namespace)

**Namespaces** `http://xml.juniper.net/junos/commit-scripts/1.0`

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

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

**Release Information** Function introduced in Junos OS Release 9.3.

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

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

*rpc*—Remote procedure call (RPC) to execute.

**Return Value** *result*—Results of the executed RPC, which include the contents of the **<rpc-reply>** element, but not the **<rpc-reply>** tag itself. This **\$result** variable is the same as that

produced by the `jcs:invoke()` function. By default, the results are in XML format equivalent to the output produced with the `| display xml` option in the CLI.

**Usage Examples** In the following example, the `$rpc` variable is declared and initialized with the Junos XML `<get-interface-information>` element. A call to the `jcs:open()` function generates a connection handle to the remote device at IP address 10.10.10.1. The user's login and password are provided as arguments to `jcs:open()` to provide access to the remote device. The code calls `jcs:execute()` and passes in the connection handle and RPC as arguments. Junos OS on the remote device processes the RPC and returns the results, which are stored in the `$results` variable.

```
var $rpc = <get-interface-information>;
var $connection = jcs:open('10.10.10.1', 'bsmith', 'test123');
var $results = jcs:execute($connection, $rpc);
expr $results;
```

The equivalent XSLT code is:

```
<xsl:variable name="connection" select="jcs:open('10.10.10.1', 'bsmith', 'test123')"/>
<xsl:variable name="rpc">
  <get-interface-information/>
</xsl:variable>
<xsl:variable name="results" select="jcs:execute($connection, $rpc)"/>
<xsl:value-of select="$results"/>
```

### first-of() Function (jcs and slax Namespaces)

<b>Namespaces</b>	<code>http://xml.juniper.net/junos/commit-scripts/1.0</code> <code>http://xml.libslax.org/slax</code>
<b>SLAX Syntax</b>	<code>var \$result = prefix:first-of(object, "expression");</code>
<b>XSLT Syntax</b>	<code>&lt;xsl:variable name="result" select="prefix:first-of(object, 'expression')"/&gt;</code>
<b>Release Information</b>	Function introduced in Junos OS Release 7.6. Support for the slax namespace <code>http://xml.libslax.org/slax</code> added in Junos OS Release 12.2.
<b>Description</b>	Return the first nonempty (non-null) item in a list. If all objects in the list are empty, the default expression is returned. This function provides the same functionality as an <code>if / else-if / else</code> construct but in a much more concise format.  The <i>prefix</i> associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.
<b>Parameters</b>	<i>expression</i> —Default value returned if all objects in the list are empty.  <i>object</i> —List of objects.
<b>Return Value</b>	<i>result</i> —First nonempty (non-null) item in the object list. If all objects in the list are empty, the default expression is returned.

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

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

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

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

The equivalent XSLT code is:

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

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

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

See also *Example: Displaying DNS Hostname Information Using an Op Script*.

## get-command() Function (jcs and slax Namespaces)

**Namespaces** <http://xml.juniper.net/junos/commit-scripts/1.0>  
<http://xml.libslax.org/slax>

**SLAX Syntax** *string* = *prefix*:get-command(*string*);

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

<b>Description</b>	Prompt the user for input and return the input as a string. If the readline (or libedit) library was found at install time, the return string is entered in the readline history, and will be available using the readline history keystrokes (Ctrl+P and Ctrl+N).  The <i>prefix</i> associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.
<b>Parameters</b>	<i>string</i> —Prompt text.
<b>Return Value</b>	<i>string</i> —Command text entered by the user.
<b>Usage Examples</b>	<pre>var \$response = slax:get-command("# ");</pre>

### get-hello() Function (jcs Namespace)

<b>Namespaces</b>	<a href="http://xml.juniper.net/junos/commit-scripts/1.0">http://xml.juniper.net/junos/commit-scripts/1.0</a>
<b>SLAX Syntax</b>	<pre>var \$capabilities = jcs:get-hello(<i>connection</i>);</pre>
<b>XSLT Syntax</b>	<pre>&lt;xsl:variable name="capabilities" select="jcs:get-hello(<i>connection</i>)"/&gt;</pre>
<b>Release Information</b>	Function introduced in Junos OS Release 11.4.
<b>Description</b>	<p>Return the session ID and the capabilities of the NETCONF server during a NETCONF session.</p> <p>During session establishment, the NETCONF server and client application each emit a <b>&lt;hello&gt;</b> element to specify which operations, or <i>capabilities</i>, they support from among those defined in the NETCONF specification or published as proprietary extensions. The <b>&lt;hello&gt;</b> element encloses the <b>&lt;capabilities&gt;</b> element and the <b>&lt;session-id&gt;</b> element, which specifies the session ID for this NETCONF session.</p> <p>Within the <b>&lt;capabilities&gt;</b> element, a <b>&lt;capability&gt;</b> element specifies each supported function. Each capability defined in the NETCONF specification is represented by a uniform resource name (URN). Capabilities defined by individual vendors are represented by uniform resource identifiers (URIs), which can be URNs or URLs.</p>
<b>Parameters</b>	<i>connection</i> —Connection handle generated by a call to the <b>jcs:open()</b> function.
<b>Return Value</b>	<i>capabilities</i> —XML node set that specifies which operations, or <i>capabilities</i> , the NETCONF server supports. The node set also includes the session ID.
<b>Usage Examples</b>	<p>In the following code snippet, the user, bsmith, establishes a NETCONF session on the default port with the remote device, fivestar, which is running Junos OS. Since the code does not specify a value for the password, the user is prompted for a password during script execution. Once authentication is established, the code calls the <b>jcs:get-hello()</b> function and stores the return value in the variable <b>\$hello</b>, which is then printed to the CLI.</p> <pre>var \$netconf := {   &lt;method&gt; "netconf";</pre>

```

    <username> "bsmith";
  }
  var $connection = jcs:open("fivestar", $netconf);
  var $hello = jcs:get-hello($connection);
  expr jcs:output($hello);
  expr jcs:close($connection);

```

The CLI displays the following output:

```

bsmith@fivestar's password:

urn:ietf:params:xml:ns:netconf:base:1.0
urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
urn:ietf:params:xml:ns:netconf:capability:validate:1.0
urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
http://xml.juniper.net/netconf/junos/1.0
http://xml.juniper.net/dmi/system/1.0

20847

```

### get-input() Function (jcs and slax Namespaces)

<b>Namespaces</b>	http://xml.juniper.net/junos/commit-scripts/1.0 http://xml.libslax.org/slax
<b>SLAX Syntax</b>	var \$user-input = <i>prefix</i> :get-input( <i>string</i> );
<b>XSLT Syntax</b>	<xsl:variable name="user-input" select=" <i>prefix</i> :get-input( <i>string</i> )"/>
<b>Release Information</b>	Function introduced in Junos OS Release 9.4. Support for the slax namespace <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a> added in Junos OS Release 12.2.
<b>Description</b>	Invoke a CLI prompt and wait for user input. The user input is defined as a string for subsequent use. If the script is run non-interactively, the function returns an empty value. This function cannot be used with event scripts.  The <i>prefix</i> associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.
<b>Parameters</b>	<i>string</i> —CLI prompt text.
<b>Return Value</b>	user-input—Text typed by the user and stored as a string. The return value will be empty if the script is run non-interactively.
<b>Usage Examples</b>	In the following example, the user is prompted to enter a login name. The user's input is stored in the variable <code>\$username</code> .  <pre>var \$username = jcs:get-input("Enter login id: ");</pre>

### get-protocol() Function (jcs Namespace)

**Namespaces**    <http://xml.juniper.net/junos/commit-scripts/1.0>

<b>SLAX Syntax</b>	<code>var \$protocol = jcs:get-protocol(connection);</code>
<b>XSLT Syntax</b>	<code>&lt;xsl:variable name="protocol" select="jcs:get-protocol(connection)"/&gt;</code>
<b>Release Information</b>	Function introduced in Junos OS Release 11.4.
<b>Description</b>	Return the session protocol associated with the connection handle. The protocol values are <code>junoscript</code> , <code>netconf</code> , and <code>junos-netconf</code> .
<b>Parameters</b>	<i>connection</i> —Connection handle generated by a call to the <code>jcs:open()</code> function.
<b>Return Value</b>	<i>protocol</i> —Session protocol associated with the connection handle. The values are <code>junoscript</code> , <code>netconf</code> , and <code>junos-netconf</code> .
<b>Usage Examples</b>	In the following code snippet, the user, bsmith, establishes a NETCONF session on the default port with the remote device, fivestar. Since the code does not specify a value for the password, the user is prompted for a password during script execution. Once authentication is established, the code calls the <code>jcs:get-protocol()</code> function and stores the return value in the variable <code>\$protocol</code> , which is then printed to the CLI.

```
var $netconf := {  
  <method> "netconf";  
  <username> "bsmith";  
}  
var $connection = jcs:open("fivestar", $netconf);  
var $protocol = jcs:get-protocol($connection);  
expr jcs:output($protocol);  
expr jcs:close($connection);
```

The CLI displays the following output:

```
bsmith@fivestar's password:  
  
netconf
```

### get-secret() Function (jcs and slax Namespaces)

<b>Namespaces</b>	<code>http://xml.juniper.net/junos/commit-scripts/1.0</code> <code>http://xml.libslax.org/slax</code>
<b>SLAX Syntax</b>	<code>var \$user-input = prefix:get-secret(string);</code>
<b>XSLT Syntax</b>	<code>&lt;xsl:variable name="user-input" select="prefix:get-secret(string)"/&gt;</code>
<b>Release Information</b>	Function introduced in Junos OS Release 9.5R2. Support for the slax namespace <code>http://xml.libslax.org/slax</code> added in Junos OS Release 12.2.
<b>Description</b>	Invoke a CLI prompt and wait for user input. Unlike the <code>jcs:get-input()</code> function, the input is not echoed back to the user, which makes the function useful for obtaining passwords. The user input is defined as a string for subsequent use. This function cannot be used with event scripts.



The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

- Parameters** *string*—CLI prompt text.
- Return Value** *user-input*—Text typed by the user and stored as a string.
- Usage Examples** The following example shows how to prompt for a password that is not echoed back to the user:

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

### hostname() Function (jcs Namespace)

- Namespaces** <http://xml.juniper.net/junos/commit-scripts/1.0>
- SLAX Syntax** `var $name = jcs:hostname(expression);`
- XSLT Syntax** `<xsl:variable name="name" select="jcs:hostname(expression)"/>`
- Release Information** Function introduced in Junos OS Release 9.3.
- Description** Return the fully qualified domain name associated with a given IPv4 or IPv6 address. The DNS server must be configured on the device in order to resolve the domain name.
- Parameters** *expression*—IPv4 or IPv6 address.
- Return Value** *name*—Hostname associated with the IP address.
- Usage Examples** The following example initializes the variable **address** with the IP address 10.10.10.1. The **\$address** variable is passed as the argument to the **jcs:hostname()** function. If the DNS server is configured on the device, the function will resolve the IP address and return the fully qualified domain name, which is stored in the variable **host**.

```
var $address = "10.10.10.1";  
var $host = jcs:hostname($address);
```

In XSLT:

```
<xsl:variable name="address" select="10.10.10.1">  
<xsl:variable name="host" select="jcs:hostname($address)"/>
```

### invoke() Function (jcs Namespace)

- Namespaces** <http://xml.juniper.net/junos/commit-scripts/1.0>
- SLAX Syntax** `var $result = jcs:invoke(rpc);`
- XSLT Syntax** `<xsl:variable name="result" select="jcs:invoke(rpc)"/>`
- Release Information** Function introduced in Junos OS Release 7.6.

<b>Description</b>	Invoke a remote procedure call (RPC) on the local device. The function is called with one argument, either a string containing a Junos XML API RPC, or a tree containing an RPC. The result contains the contents of the <b>&lt;rpc-reply&gt;</b> element, not including the <b>&lt;rpc-reply&gt;</b> tag. An RPC allows you to perform functions equivalent to Junos OS operational mode commands.
<b>Parameters</b>	<i>rpc</i> —String containing a Junos XML API RPC or a tree containing an RPC.
<b>Return Value</b>	<i>result</i> —Results of the executed RPC, which include the contents of the <b>&lt;rpc-reply&gt;</b> element, but not the <b>&lt;rpc-reply&gt;</b> tag itself. By default, the results are in XML format equivalent to the output produced with the <b>  display xml</b> option in the CLI.
<b>Usage Examples</b>	The following example tests to see if the <b>interface</b> argument is included on the command line when the script is executed. If the argument is provided, the output of the <b>show interfaces terse</b> operational mode command is narrowed to include only information about the specified interface.

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

In this example, the **jcs:invoke()** function calls the Junos XML API RPC **get-software-information**, and stores the unmodified output in the variable **sw**:

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

## open() Function (jcs Namespace)

<b>Namespaces</b>	<a href="http://xml.juniper.net/junos/commit-scripts/1.0">http://xml.juniper.net/junos/commit-scripts/1.0</a>
<b>SLAX Syntax</b>	<pre>var \$connection = jcs:open(); var \$connection = jcs:open(<i>remote-hostname</i>, &lt;username&gt;, &lt;passphrase&gt;,   &lt;routing-instance-name&gt;); var \$connection = jcs:open(<i>remote-hostname</i>, &lt;session-options&gt;);</pre>
<b>XSLT Syntax</b>	<pre>&lt;xsl:variable name="connection" select="jcs:open()"/&gt; &lt;xsl:variable name="connection" select="jcs:open(<i>remote-hostname</i>, &lt;username&gt;,   &lt;passphrase&gt;, &lt;routing-instance-name&gt;)/&gt; &lt;xsl:variable name="connection" select="jcs:open(<i>remote-hostname</i>, &lt;session-options&gt;)/&gt;</pre>
<b>Release Information</b>	Function introduced in Junos OS Release 9.3. Support for NETCONF sessions added in Junos OS Release 11.4R1. Support for routing instances added in Junos OS Release 12.2R1.

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

To redirect the SSH connection to originate from within a specific routing instance, include the routing instance name in the connection parameters. The routing instance must be configured at the **[edit routing-instances]** hierarchy level, and the remote device must be reachable either using the routing table for that routing instance or from one of the interfaces configured under that routing instance.

Starting with Junos OS Release 11.4, the new parameter, *session-options*, supports the option to create a session either with the Junos XML protocol server on devices running Junos OS or with the NETCONF server on devices where NETCONF service over SSH is enabled. Previously, the function supported only sessions with the Junos XML protocol server on devices running Junos OS.

The connection handle is closed with the `jcs:close()` function.

**Parameters** *passphrase*—(Optional) User's login passphrase. If you do not specify a passphrase and it is required for authentication, you should be prompted for one during script execution by the device to which you are connecting.

*remote-hostname*—Domain name or IP address of the remote router, switch, or security device. If you are opening a local connection, do not pass this value. If you specify a session type, this parameter is required.

*routing-instance-name*—(Optional) Routing instance from within which the SSH connection originates.

*session-options*—(Optional) XML node set that specifies the session protocol and connection parameters. The structure of the node set is:

```
var $session-options := {  
  <instance> "routing-instance-name";  
  <method> ("junoscript" | "netconf" | "junos-netconf");  
  <passphrase> "passphrase";  
  <password> "password";  
  <port> "port-number";  
  <routing-instance> "routing-instance-name";  
  <username> "username";  
}
```

- **<instance>**—(Optional) Routing instance from within which the SSH connection originates. This element is identical to **<routing-instance>**.
- **<method>**—(Optional) Session protocol. The protocol is one of three values: **junoscript**, **netconf**, or **junos-netconf**. If you do not specify a protocol, a **junoscript** session is created by default. A **<method>** value of **junoscript** establishes a session with the Junos XML protocol server on a device running Junos OS. A **<method>** value of **netconf** establishes a session with a NETCONF server over an SSHv2

connection. A **<method>** value of **junos-netconf** establishes a session with a NETCONF server over an SSHv2 connection on a device running Junos OS.

- **<passphrase> or <password>**—(Optional) User's login passphrase. If you do not specify a passphrase and it is required for authentication, you should be prompted for one during script execution by the device to which you are connecting.
- **<port>**—(Optional) Server port number for **netconf** and **junos-netconf** sessions. For NETCONF sessions, **jcs:open()** connects to the NETCONF server at the default port 830. If you specify a value for **<port>**, **jcs:open()** connects to the given port instead. Specifying a port number has no impact on **junoscript** sessions, which are always established over SSH port 22.
- **<routing-instance>**—(Optional) Routing instance from within which the SSH connection originates. This element is identical to **<instance>**.
- **<username>**—(Optional) User's login name. If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script.

*username*—(Optional) User's login name. If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script.

**Return Value** connection—Connection handle to the remote host.

**Usage Examples** The following example shows how to connect to a local device:

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

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

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

The following example shows how the user, bsmith, with the passphrase "test123" obtains a connection handle to the remote device, fivestar:

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

The following example shows how the user, bsmith, with the passphrase "test123" creates a **junos-netconf** session with a device running Junos OS:

```
var $options := {  
  <method> "junos-netconf";  
  <username> "bsmith";  
  <passphrase> "test123";  
}  
var $connection = jcs:open("fivestar", $options);
```

## output() Function (jcs and slax Namespaces)

**Namespaces** <http://xml.juniper.net/junos/commit-scripts/1.0>  
<http://xml.libslax.org/slax>

**SLAX Syntax** `expr prefix:output('string', <string>);`

**XSLT Syntax** `<xsl:value-of select="prefix:output('string', <string>)" />`

- Release Information** Function introduced in Junos OS Release 7.6.  
Support for the slax namespace <http://xml.libslax.org/slax> added in Junos OS Release 12.2.
- Description** Display one or more lines of output text, either to the CLI user (when used in op scripts), or to the output file (when used in event scripts). The function can be called with either a single string argument or with multiple string arguments. Multiple arguments are concatenated into one combined string. A newline terminates the output text.

`jcs:output()` is not supported in commit scripts. Commit scripts use the `<xnm:warning>` and `<xnm:error>` result tree elements to display text to the CLI user.

The behavior of `jcs:output()` differs from the `<output>` result tree element in that `jcs:output()` displays its text immediately, rather than waiting until the conclusion of the script. This makes it suitable for scripts where user interaction is required, such as when the `jcs:get-input()` function is used, or when status messages should be displayed in the midst of script processing. While `jcs:output()` does return a node set, it is always empty and can be ignored. Therefore, the `jcs:output()` function is normally called with the `expr` statement, rather than assigning its result to a variable.

The following escape characters are supported in the output text:

- `\\` –Backslash (as of Junos OS Release 10.2)
- `\r` –Carriage Return
- `\"` –Double-quote (as of Junos OS Release 10.1R2)
- `\n` –Newline
- `\'` –Single-quote
- `\t` –Tab

Starting with Junos OS Release 10.2, the maximum length for output text is 10 KB, and longer strings are truncated to the supported length.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

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

- Usage Examples** SLAX syntax:

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

XSLT syntax:

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

### parse-ip() Function (jcs Namespace)

- Namespaces** <http://xml.juniper.net/junos/commit-scripts/1.0>
- SLAX Syntax** `var $result = jcs:parse-ip("ipaddress/prefix-length | netmask");`

<b>XSLT Syntax</b>	<code>&lt;xsl:variable name="result" select="jcs:parse-ip('ipaddress/(prefix-length   netmask)')"/&gt;</code>
<b>Release Information</b>	Function introduced in Junos OS Release 9.0.
<b>Description</b>	Parse an IPv4 or IPv6 address.
<b>Parameters</b>	<p><i>ipaddress</i>—IPv4 or IPv6 address.</p> <p><i>netmask</i>—Netmask defining the network prefix portion of the address.</p> <p><i>prefix-length</i>—Prefix length defining the number of bits used in the network prefix portion of the address.</p>
<b>Return Value</b>	<p>result—An array containing:</p> <ul style="list-style-type: none"><li>• Host IP address (or <b>NULL</b> in the case of an error)</li><li>• Protocol family (inet for IPv4 or inet6 for IPv6)</li><li>• Prefix length</li><li>• Network address</li><li>• Network mask in dotted decimal notation for IPv4 addresses (left blank for IPv6 addresses)</li></ul>
<b>Usage Examples</b>	<p>In the following examples, an IPv4 address and an IPv6 address are parsed and the resulting output is detailed:</p> <pre>var \$addr = jcs:parse-ip("10.1.2.10/255.255.255.0");</pre> <ul style="list-style-type: none"><li>• <b>\$addr[1]</b> contains the host address <b>10.1.2.10</b>.</li><li>• <b>\$addr[2]</b> contains the protocol family <b>inet</b>.</li><li>• <b>\$addr[3]</b> contains the prefix length <b>24</b>.</li><li>• <b>\$addr[4]</b> contains the network address <b>10.1.2.0</b>.</li><li>• <b>\$addr[5]</b> contains the netmask for IPv4 <b>255.255.255.0</b>.</li></ul> <pre>var \$addr = jcs:parse-ip("2001:DB8::c50:8a:800:200C:417A/32");</pre> <ul style="list-style-type: none"><li>• <b>\$addr[1]</b> contains the host address <b>2001:db8:0:c50:8a:800:200c:417a</b>.</li><li>• <b>\$addr[2]</b> contains the protocol family <b>inet6</b>.</li><li>• <b>\$addr[3]</b> contains the prefix length <b>32</b>.</li><li>• <b>\$addr[4]</b> contains the network address <b>2001:db8::</b>.</li><li>• <b>\$addr[5]</b> is blank for IPv6 (<b>""</b>).</li></ul>

## printf() Function (jcs and slax Namespaces)

<b>Namespaces</b>	<a href="http://xml.juniper.net/junos/commit-scripts/1.0">http://xml.juniper.net/junos/commit-scripts/1.0</a> <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a>
-------------------	--

<b>SLAX Syntax</b>	<code>expr <i>prefix</i>:printf(<i>expression</i>);</code>
<b>XSLT Syntax</b>	<code>&lt;xsl:value-of select="<i>prefix</i>:printf(<i>expression</i>)"/&gt;</code>
<b>Release Information</b>	Function introduced in Junos OS Release 7.6. Support for the slax namespace <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a> added in Junos OS Release 12.2.
<b>Description</b>	<p>Generate formatted output text. Most standard <b>printf</b> formats are supported, in addition to some Junos OS–specific formats. The function returns a formatted string but does not print it on call. To use the following Junos OS modifiers, place the modifier between the percent sign (%) and the conversion specifier.</p> <ul style="list-style-type: none"> <li>• <b>j1</b>—Operator that emits the field only if it changed from the last time the function was called. This assumes that the expression's format string is unchanged.</li> <li>• <b>jc</b>—Operator that capitalizes the first letter of the associated output string.</li> <li>• <b>jt{TAG}</b>—Operator that emits the tag if the associated argument is not empty.</li> </ul> <p>The <i>prefix</i> associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.</p>
<b>Parameters</b>	<i>expression</i> —Format string containing an arbitrary number of format specifiers and associated arguments to output.
<b>Usage Examples</b>	<p>In the following example, the <b>j1</b> operator suppresses printing the interface identifier so-0/0/0 in the second line of output, because the identifier argument has not changed from the first printing. The <b>jc</b> operator capitalizes the output strings <b>up</b> and <b>down</b>. The <b>jt{--}</b> operator does not print the <b>{--}</b> tag in the first line of output, because the associated output argument is an empty string. However, the tag is printed in the second line because the associated output is the non-empty string <b>test</b>.</p>

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

produces the following output:

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

### progress() Function (jcs and slax Namespaces)

<b>Namespaces</b>	<a href="http://xml.juniper.net/junos/commit-scripts/1.0">http://xml.juniper.net/junos/commit-scripts/1.0</a> <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a>
<b>SLAX Syntax</b>	<code>expr <i>prefix</i>:progress('string');</code>
<b>XSLT Syntax</b>	<code>&lt;xsl:value-of select="<i>prefix</i>:progress('string')"/&gt;</code>
<b>Release Information</b>	Function introduced in Junos OS Release 7.6.

Support for the slax namespace <http://xml.libslax.org/slax> added in Junos OS Release 12.2.

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

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

**Parameters** *string*—Text output to CLI session

**Usage Examples** SLAX syntax:

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

XSLT syntax:

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

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

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

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

## regex() Function (jcs and slax Namespaces)

**Namespaces** <http://xml.juniper.net/junos/commit-scripts/1.0>  
<http://xml.libslax.org/slax>

**SLAX Syntax** `var $result = prefix:regex(pattern, string);`

**XSLT Syntax** `<xsl:variable name="result" select="prefix:regex(pattern, string)"/>`

**Release Information** Function introduced in Junos OS Release 7.6  
Support for the slax namespace <http://xml.libslax.org/slax> added in Junos OS Release 12.2.

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

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

**Parameters** *pattern*—Regular expression that is evaluated against the string argument.

*string*—String within which to search for matches of the specified regular expression.



**Return Value** result—Array of strings that match the given regex pattern within the string argument.

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

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

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

### sleep() Function (jcs and slax Namespaces)

**Namespaces** <http://xml.juniper.net/junos/commit-scripts/1.0>  
<http://xml.libslax.org/slax>

**SLAX Syntax** `expr prefix:sleep(seconds, <milliseconds>);`

**XSLT Syntax** `<xsl:value-of select="prefix:sleep(seconds, <milliseconds>)" />`

**Release Information** Function introduced in Junos OS Release 7.6.  
 Support for the slax namespace <http://xml.libslax.org/slax> added in Junos OS Release 12.2.

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

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

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

*seconds*—Number of seconds the script should sleep.

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

SLAX syntax:

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

XSLT syntax:

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

## split() Function (jcs and slax Namespaces)

<b>Namespaces</b>	<a href="http://xml.juniper.net/junos/commit-scripts/1.0">http://xml.juniper.net/junos/commit-scripts/1.0</a> <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a>
<b>SLAX Syntax</b>	<code>var \$substrings = <i>prefix</i>:split(<i>expression</i>, <i>string</i>, &lt;<i>limit</i>&gt;);</code>
<b>XSLT Syntax</b>	<code>&lt;xsl:variable name="substrings" select="<i>prefix</i>:split(<i>expression</i>, <i>string</i>, &lt;<i>limit</i>&gt;)" /&gt;</code>
<b>Release Information</b>	Function introduced in Junos OS Release 8.4 Support for the slax namespace <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a> added in Junos OS Release 12.2.
<b>Description</b>	<p>Split a string into an array of substrings delimited by a regular expression pattern. If the optional integer argument <i>limit</i> is specified, the function splits the entire string into <i>limit</i> number of substrings. If there are more than <i>limit</i> number of matches, the substrings include the first <i>limit</i>-1 matches as well as the remaining portion of the original string for the last match.</p> <p>The <i>prefix</i> associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.</p>
<b>Parameters</b>	<p><i>expression</i>—Regular expression pattern used as the delimiter.</p> <p><i>limit</i>—(Optional) Number of substrings into which to break the original string.</p> <p><i>string</i>—Original string.</p>
<b>Return Value</b>	<i>\$substrings</i> —Array of <i>limit</i> number of substrings. If <i>limit</i> is not specified, the result array size is equal to the number of substrings extracted from the original string as determined by the specified delimiter.
<b>Usage Examples</b>	<p>In the following example, the original string is "123:abc:456:xyz:789". The <code>jcs:split()</code> function breaks this string into substrings that are delimited by the regular expression pattern, which in this case is a colon(:). The optional parameter <i>limit</i> is not specified, so the function returns an array containing all the substrings that are bounded by the delimiter(:).</p> <pre>var \$pattern = "(:)"; var \$substrings = jcs:split(\$pattern, "123:abc:456:xyz:789");</pre> <p>returns:</p> <pre>\$substrings[1] == "123" \$substrings[2] == "abc"</pre>

```
$substrings[3] == "456"
$substrings[4] == "xyz"
$substrings[5] == "789"
```

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

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

returns:

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

### sysctl() Function (jcs and slax Namespaces)

<b>Namespaces</b>	<a href="http://xml.juniper.net/junos/commit-scripts/1.0">http://xml.juniper.net/junos/commit-scripts/1.0</a> <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a>
<b>SLAX Syntax</b>	var \$value = <i>prefix</i> :sysctl( <i>sysctl-value</i> , "(i   s)");
<b>XSLT Syntax</b>	<xsl:variable name="value" select=" <i>prefix</i> :sysctl( <i>sysctl-value</i> , '(i   s)')"/>
<b>Release Information</b>	Function introduced in Junos OS Release 7.6 Support for the slax namespace <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a> added in Junos OS Release 12.2.
<b>Description</b>	Return the given <b>sysctl</b> value as a string or an integer. Use the "i" argument to specify an integer. Use the "s" argument to specify a string.  The <i>prefix</i> associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.
<b>Parameters</b>	<i>sysctl-value</i> — <b>sysctl</b> value to convert to a string or integer.
<b>Return Value</b>	\$value—Returned string or integer value.
<b>Usage Examples</b>	var \$value = jcs:sysctl("kern.hostname", "s");

### syslog() Function (jcs and slax Namespaces)

<b>Namespaces</b>	<a href="http://xml.juniper.net/junos/commit-scripts/1.0">http://xml.juniper.net/junos/commit-scripts/1.0</a> <a href="http://xml.libslax.org/slax">http://xml.libslax.org/slax</a>
<b>SLAX Syntax</b>	expr <i>prefix</i> :syslog( <i>priority</i> , <i>message</i> , < <i>message2</i> >, < <i>message3</i> > ...);
<b>XSLT Syntax</b>	<xsl:value-of select=" <i>prefix</i> :syslog( <i>priority</i> , <i>message</i> , < <i>message2</i> >, < <i>message3</i> >)"/>
<b>Release Information</b>	Function introduced in Junos OS Release 7.6

Support for the slax namespace <http://xml.libslax.org/slax> added in Junos OS Release 12.2.

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

The **prefix** associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

**Parameters** *message*—String or variable that is output to the system log file.

*message2*—(Optional) Any additional number of strings or variable names passed as arguments to the function. These are concatenated with the **message** argument and output to the system log file.

*priority*—Priority given to the syslog message. The priority can be specified as a **facility.severity** string, or it can be expressed as an integer calculated from the corresponding numeric values of the facility and severity strings. [Table 5 on page 38](#) and [Table 6 on page 39](#) show the facility and severity strings available and their corresponding numeric values.

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

**Table 5: Facility Strings**

Facility String	Description	Numeric Value
auth	Authorization system	4
change	Configuration change log	22
conflict	Configuration conflict log	21
daemon	Various system processes	3
external	Local external applications	18
firewall	Firewall filtering system	19
ftp	FTP processes	11
interact	Commands executed by the UI	23
pfe	Packet Forwarding Engine	20
user	User processes	1

Table 6: Severity Strings

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

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

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

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

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

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

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

### trace() Function (jcs and slax Namespaces)

**Namespaces** <http://xml.juniper.net/junos/commit-scripts/1.0>  
<http://xml.libslax.org/slax>

**SLAX Syntax** `expr prefix:trace('expression');`

**XSLT Syntax** `<xsl:value-of select="prefix:trace('expression')"/>`

**Release Information** Function introduced in Junos OS Release 7.6.  
 Support for the slax namespace <http://xml.libslax.org/slax> added in Junos OS Release 12.2.

**Description** Issue a trace message, which is sent to the trace file. You must configure **traceoptions** under the respective script type in the configuration hierarchy in order to output a message to the trace file using the **jcs:trace()** function. The output goes to the configured trace

file. If **traceoptions** is enabled, but no trace file is explicitly configured, the output goes to the default trace file for that script type.

The *prefix* associated with the namespace URI should be defined in the prefix-to-namespace mapping in the style sheet.

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

**Usage Examples** SLAX syntax:

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

XSLT syntax:

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

---

## Junos Script Automation: Named Templates in the jcs Namespace Overview

Junos OS provides several named templates to make scripting tasks easier in commit, op, and event scripts. The named templates reside in the **junos.xml** import file, which is included with the standard Junos OS installation available on all switches, routers, and security devices running Junos OS.

The Junos OS named templates are defined in the namespace with the associated Uniform Resource Identifier (URI) <http://xml.juniper.net/junos/commit-scripts/1.0>. The templates use the **jcs:** prefix to avoid conflicting with standard XSLT templates or user-defined templates of the same name in a script. To use the Junos named templates in a script, you must include the namespace URI in your style sheet declaration. Map the **jcs** prefix to the URI by including the **xmlns:jcs** attribute in the opening **<xsl:stylesheet>** tag element for XSLT scripts or by including the **ns jcs** statement in SLAX scripts. You must also import the **junos.xml** file into the script by including the **<xsl:import/>** tag element in XSLT scripts or the **import** statement in SLAX scripts and specifying the **junos.xml** file location.

To call a named template in a script, include the **<xsl:call-template name="template-name">** element for XSLT scripts or the **call** statement for SLAX scripts and pass along any required or optional parameters. Template parameters are assigned by name and can appear in any order. This differs from functions where the arguments must be passed into the function in the precise order specified by the function definition.

The following example imports the **junos.xml** file into a script and maps the **jcs** prefix to the namespace identified by the URI <http://xml.juniper.net/junos/commit-scripts/1.0>. The script demonstrates a call to the **jcs:edit-path** template.

**XSLT Syntax**

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
  <xsl:import href="../../import/junos.xml"/>
  ...
  <xsl:for-each select="interfaces/interface[starts-with(name, 'so-')]">
    <xnm:warning>
      <xsl:call-template name="jcs:edit-path"/>
    </xnm:warning>
  </xsl:for-each>
</xsl:stylesheet>
```

```

        <message>interface configured</message>
    </xnm:warning>
</xsl:for-each>
...
</xsl:stylesheet>

```

**SLAX Syntax**

```

version 1.0;
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xml";
...
for-each ( interfaces/interface[starts-with(name,'so-') ] ) {
    <xnm:warning> {
        call jcs:edit-path();
        <message> "interface configured";
    }
}
...

```

For more information about attributes and tag elements to include in your scripts, see *Required Boilerplate for Commit Scripts*, *Required Boilerplate for Op Scripts*, and *Required Boilerplate for Event Scripts*.

**Related Documentation**

- [Summary of Extension Functions in the jcs and slax Namespaces on page 13](#)
- [Junos Named Templates in the jcs Namespace Summary on page 41](#)
- [Junos Script Automation: Understanding Extension Functions in the jcs and slax Namespaces on page 11](#)
- [Junos Script Automation: Global Parameters and Variables in the junos.xml File on page 51](#)

## Junos Named Templates in the jcs Namespace Summary

The Junos named templates are summarized in the following table:

**Table 7: Junos Named Templates**

Template	Description
<a href="#">jcs:edit-path</a>	Generate an <b>&lt;edit-path&gt;</b> element suitable for inclusion in an <b>&lt;xnm:error&gt;</b> or <b>&lt;xnm:warning&gt;</b> element.
<a href="#">jcs:emit-change</a>	Generate a <b>&lt;change&gt;</b> or <b>&lt;transient-change&gt;</b> element, which results in a persistent or transient change to the configuration.
<a href="#">jcs:emit-comment</a>	Emit a simple comment that indicates a change was made by a commit script.
<a href="#">jcs:grep</a>	Search a file for all instances matching a specified regular expression and write the matching strings and corresponding lines to the result tree.

Table 7: Junos Named Templates (*continued*)

Template	Description
<a href="#">jcs:load-configuration</a>	Make structured changes to the Junos OS configuration using an op script.
<a href="#">jcs:statement</a>	Generate a <b>&lt;statement&gt;</b> element suitable for inclusion in an <b>&lt;xnm:error&gt;</b> or <b>&lt;xnm:warning&gt;</b> element.

**Related Documentation**

- [Junos Script Automation: Understanding Extension Functions in the jcs and slax Namespaces on page 11](#)
- [Junos Script Automation: Named Templates in the jcs Namespace Overview on page 40](#)
- [Junos Script Automation: Global Parameters and Variables in the junos.xml File on page 51](#)

## Junos Named Templates in the jcs Namespace

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

- [jcs:edit-path Template on page 42](#)
- [jcs:emit-change Template on page 43](#)
- [jcs:emit-comment Template on page 45](#)
- [jcs:grep Template on page 46](#)
- [jcs:load-configuration Template on page 47](#)
- [jcs:statement Template on page 50](#)

### jcs:edit-path Template

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

**SLAX Syntax**      `call jcs:edit-path($dot=expression);`

**Description**      Generate an **<edit-path>** element suitable for inclusion in an **<xnm:error>** or **<xnm:warning>** element. This template converts a location in the configuration hierarchy into the standard text representation that you would see in the Junos OS configuration mode banner. By default, the location of the configuration error is passed into the **jcs:edit-path** template as the value of **dot**. This location defaults to ".", the current position in the XML hierarchy. You can alter the default by including a valid XPath expression for the **dot** parameter when you call the template.

**Parameters**      **dot**—XPath expression specifying the hierarchy level. The default location is the position in the XML hierarchy that the script is currently evaluating. You can alter the default when you call the template by including a valid XPath expression either for the **dot** parameter in SLAX scripts or for the **select** attribute of the **dot** parameter in XSLT scripts.



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

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

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

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

## jcs:emit-change Template

<b>XSLT Syntax</b>	<pre>&lt;xsl:call-template name="jcs:emit-change"&gt;   &lt;xsl:with-param name="content"&gt;     ...   &lt;/xsl:with-param&gt;   &lt;xsl:with-param name="dot" select="expression"/&gt;   &lt;xsl:with-param name="message"&gt;     &lt;xsl:text&gt;message&lt;/xsl:text&gt;   &lt;/xsl:with-param&gt;   &lt;xsl:with-param name="name" select="name(\$dot)"/&gt;   &lt;xsl:with-param name="tag" select="(change   transient-change)"/&gt; &lt;/xsl:call-template&gt;</pre>
<b>SLAX Syntax</b>	<pre>call jcs:emit-change(\$dot=expression, \$name = name(\$dot), \$tag = "(change   transient-change)" {   with \$content = {     ...   }   with \$message = {     expr "message";   } }</pre>

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

**Parameters** This template includes the following optional parameters:

**content**—Content of the persistent or transient change, relative to **dot**.

**dot**—XPath expression specifying the hierarchy level at which the change will be made. The default location is the position in the XML hierarchy that the script is currently evaluating. You can alter the default when you call the template by including a valid

XPath expression either for the **dot** parameter in SLAX scripts or for the **select** attribute of the **dot** parameter in XSLT scripts.

**message**—Warning message displayed in the CLI notifying the user that the configuration has been changed. The message parameter automatically includes the edit path, which defaults to the current location in the XML hierarchy. To change the default edit path, specify a valid XPath expression either for the **dot** parameter in SLAX scripts or for the **select** attribute of the **dot** parameter in XSLT scripts.

**name**—Allows you to refer to the current element or attribute. The **name()** XPath function returns the name of an element or attribute. The **name** parameter defaults to the value **name(\$dot)**, which is the name of the element in **dot** (which in turn defaults to “.”, which is the current element).

**tag**—Type of change to generate. By default, the **jcs:emit-change** template generates a persistent change, as designated by the 'change' expression. To specify a transient change, you must include the **tag** parameter and include the 'transient-change' expression.

**Usage Examples** The following example demonstrates how to call the **jcs:emit-change** template in a commit script:

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

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

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

```
warning: Adding ISO-enabled interface so-1/3/2.0 to [protocols mpls]
commit complete
```

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

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

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

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

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

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

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

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

See also *Example: Imposing a Minimum MTU Setting*.

## jcs:emit-comment Template

XSLT Syntax	<pre>&lt;junos:comment&gt;   &lt;xsl:text&gt;...&lt;/xsl:text&gt; &lt;/junos:comment&gt;</pre>
-------------	--

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

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

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

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

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

## jcs:grep Template

**XSLT Syntax**

```
<xsl:call-template name="jcs:grep">
  <xsl:with-param name="filename" select="filename"/>
  <xsl:with-param name="pattern" select="pattern"/>
</xsl:call-template>
```

**SLAX Syntax**

```
call jcs:grep($filename=filename, $pattern=pattern);
```

**Description** Search the given input file for all instances matching the specified regular expression and write the matching strings and corresponding lines to the result tree. The pattern is matched to each line of the file. The template does not support matching a pattern spanning multiple lines.

If the regular expression contains a syntax error, the template generates an error for every line of the file. For each match, the template adds a `<match>` element, which contains `<input>` and `<output>` child tags, to the result tree. The template writes the matching string to the `<output>` element and writes the corresponding matching line to the `<input>` element.

```
<match> {
  <input>
```

```

    <output>
  }

```

Starting with Junos OS Release 11.1, if an absolute path is not specified for the input file, the default path is relative to the user's home directory for op scripts, and it is relative to the `/var/tmp/` directory for commit scripts and for event scripts that are enabled at the `[edit event-options event-script]` hierarchy level. For event scripts that are enabled at the `[edit system scripts]` hierarchy level, the default path is relative to the top-level directory, `/`.

**Parameters** `filename`—(Mandatory) Absolute or relative path and filename of the file to search.

Starting with Junos OS Release 11.1, if you do not specify an absolute path, the path is relative to the user's home directory for op scripts, and it is relative to the `/var/tmp/` directory for commit scripts and for event scripts that are enabled at the `[edit event-options event-script]` hierarchy level. For event scripts that are enabled at the `[edit system scripts]` hierarchy level, the default path is relative to the top-level directory, `/`.

`pattern`—(Mandatory) Regular expression.

**Usage Examples** *Example: Searching Files Using an Op Script*

### jcs:load-configuration Template

**XSLT Syntax**

```

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

```

**SLAX Syntax**

```

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

```

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

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

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

- Parameters**
- action**—Specifies how to load the configuration changes with respect to the candidate configuration. The following options are supported:
    - **merge**—Combine the candidate configuration and the incoming configuration changes. If the candidate configuration and the incoming configuration contain conflicting statements, the incoming statements override those in the candidate configuration.
    - **override**—Replace the entire candidate configuration.
    - **replace**—Replace existing statements in the candidate configuration with the tags of the same name that are marked with **replace:** in the incoming configuration. If there is no existing statement of the same name in the candidate configuration, the statement is added to the candidate configuration.
  - commit-options**—Node set defining options that customize the commit command. The default value is null. Supported commit options are:
    - **check**—Check the correctness of the candidate configuration syntax, but do not commit the changes.
    - **force-synchronize**—Force the commit on the other Routing Engine (ignore any warnings).
    - **log**—Write the specified message to the commit log. This is identical to the CLI configuration mode command **commit comment**.
    - **synchronize**—Synchronize the commit on both Routing Engines.
  - configuration**—XML configuration changes. The configuration changes are incorporated into the candidate configuration as specified by the action parameter. Starting with Junos OS Release 12.2, you can also supply a NULL configuration. If the configuration data value is NULL, the template performs a simple commit of the candidate configuration.
  - connection**—Connection handle generated by a call to the **jcs:open()** function.
  - rollback**—Return to a previously committed configuration. Specify the rollback number of the configuration, and the configuration you specify is loaded from the associated file. The software saves the last 50 committed configurations. The rollback parameter is available starting with Junos OS Release 12.2.

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

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

The **options** variable uses the **:=** operator to create a node-set, which is passed to the template as the value of the **commit-options** parameter. This example uses the **synchronize** commit option. If the commit succeeds, it will commit the configuration changes on both Routing Engines. The **log** tag is also included to add the description of the commit to the commit log file for future reference.

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

SLAX syntax:

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

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

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

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

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

```
<xsl:with-param name="configuration" select="$disable"/>
<xsl:with-param name="connection" select="$conn"/>
</xsl:call-template>
</xsl:variable>

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

## jcs:statement Template

<b>XSLT Syntax</b>	<pre>&lt;xsl:call-template name="jcs:statement"&gt;   &lt;xsl:with-param name="dot" select="expression"/&gt; &lt;/xsl:call-template&gt;</pre>
--------------------	---

<b>SLAX Syntax</b>	<pre>call jcs:statement(\$dot=expression);</pre>
--------------------	--

<b>Description</b>	Generate a <b>&lt;statement&gt;</b> element suitable for inclusion in an <b>&lt;xnm:error&gt;</b> or <b>&lt;xnm:warning&gt;</b> element. This location defaults to “.”, the current position in the XML hierarchy. If the error is not at the current position in the XML hierarchy, you can alter the default when you call the template by including a valid XPath expression either for the <b>dot</b> parameter in SLAX scripts or for the <b>select</b> attribute of the <b>dot</b> parameter in XSLT scripts.
--------------------	---

<b>Parameters</b>	<b>dot</b> —XPath expression specifying the hierarchy level. The default location is the position in the XML hierarchy that the script is currently evaluating. You can alter the default when you call the template by including a valid XPath expression either for the <b>dot</b> parameter in SLAX scripts or for the <b>select</b> attribute of the <b>dot</b> parameter in XSLT scripts.
-------------------	--

<b>Usage Examples</b>	The following example demonstrates how to call the <b>jcs:statement</b> template in a commit script:
-----------------------	--

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

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

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



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

## Junos Script Automation: Global Parameters and Variables in the junos.xsl File

The **junos.xsl** import file declares several predefined parameters and a global variable of type node-set, which provide information about the Junos OS environment that is useful for creating scripts that respond to a variety of complex scenarios. The global parameters and variable are available for use in any commit, op, or event script that imports the **junos.xsl** file.

To use the parameters or variable in a script, you must import the **junos.xsl** file by including the `<xsl:import>` tag in the style sheet declaration of an XSLT script or by including the **import** statement in a SLAX script and specifying the **junos.xsl** file location as shown in the following sample code:

XSLT Syntax	<pre>&lt;?xml version="1.0"?&gt; &lt;xsl:stylesheet version="1.0"&gt;   &lt;xsl:import href="../../import/junos.xsl"/&gt;   ... &lt;/xsl:stylesheet&gt;</pre>
SLAX Syntax	<pre>version 1.0; import "../../import/junos.xsl";</pre>

The default arguments are described in detail in the following sections:

- [Global Parameters on page 51](#)
- [Global Variable on page 52](#)

## Global Parameters

Several predefined global parameters are available for use in commit, op, and event scripts. The parameters provide information about the Junos OS environment. [Table 8 on page 51](#) describes the built-in arguments.

**Table 8: Predefined Parameters Available to Automation Scripts**

Name	Description	Example
\$hostname	Hostname of the local device	Tokyo
\$localtime	Local time when the script is executed	Fri Dec 10 11:42:21 2010
\$localtime-iso	Local time, in ISO format, when the script is executed	2010-12-10 11:42:21 PST
\$product	Model of the local device	m10i
\$script	Filename of the executing script	<b>test.slax</b>
\$user	Local name of the user executing the script	root

The predefined global parameters are declared in the **junos.xml** file. You do not need to declare these parameters in a script in order to use them. Access the value of the global parameters in a script by prefixing the parameter name with the dollar sign (\$), as shown in the following example:

SLAX syntax:

```
if ($user != "root") {  
    var $script-message = $user _ " does not have permission to execute " _ $script;  
    expr jcs:output($script-message);  
}
```

XSLT syntax:

```
<xsl:if test="$user != 'root'">  
    <xsl:variable name="script-message"  
        select="concat($user, ' does not have permission to execute ', $script)"/>  
    <xsl:value-of select="jcs:output($script-message)"/>  
</xsl:if>
```

## Global Variable

Starting with Junos OS Release 11.1, Junos OS also provides a single global variable, **\$junos-context**, which is accessible for use in all commit, op, or event scripts that import the **junos.xml** file. The **\$junos-context** variable is a node-set, which has elements that mirror the original global parameters described in [“Global Parameters” on page 51](#) as well as additional elements with information about the Junos OS environment, such as whether a script is executed on the master Routing Engine.

The **\$junos-context** variable contains the **<junos-context>** node and the following hierarchy, which is common to and embedded in the source tree of all scripts:

```
<junos-context>  
  <chassis></chassis>  
  <hostname></hostname>  
  <localtime></localtime>  
  <localtime-iso></localtime-iso>  
  <pid></pid>  
  <product></product>  
  <re-master/>  
  <routing-engine-name></routing-engine-name>  
  <script-type></script-type>  
  <tty></tty>  
  <user-context>  
    <class-name></class-name>  
    <login-name></login-name>  
    <uid></uid>  
    <user></user>  
  </user-context>  
</junos-context>
```

Additionally, script-specific information is available depending on the type of script executed. For op scripts, the **<op-context>** element is also included in the source tree provided to an op script:

```
<junos-context>
  <op-context>
    <via-url/>
  </op-context>
</junos-context>
```

For commit scripts, the **<commit-context>** element is also included in the source tree provided to a commit script:

```
<junos-context>
  <commit-context>
    <commit-comment>"This is a test commit"</commit-comment>
    <commit-boot/>
    <commit-check/>
    <commit-sync/>
    <commit-confirm/>
    <database-path/>
  </commit-context>
</junos-context>
```

[Table 9 on page 54](#) identifies each node of the **\$junos-context** variable node-set, provides a brief description of the node, and gives examples of values for any elements that are not input to a script as an empty tag.

Table 9: Global Variable \$junos-context Available to Automation Scripts

Parent Node	Node	Description	Example content
<junos-context>	<chassis>	Specifies whether the script is executed on a component of a routing matrix, the Root System Domain (RSD), or a Protected System Domain (PSD)	scc, lcc (TX Matrix) psd, rsd (JCS) others
	<hostname>	Hostname of the local device	Tokyo
	<localtime>	Local time when the script is executed	Fri Dec 10 11:42:21 2010
	<localtime-iso>	Local time, in ISO format, when the script is executed	2010-12-10 11:42:21 PST
	<pid>	cscript process ID	5257
	<product>	Model of the local device	m10i
	<re-master/>	Empty element included if the script is executed on the master Routing Engine	
	<routing-engine-name>	Routing Engine on which the script is executed	re0
	<tty>	TTY of the user's session	/dev/ttyl
	<script-type>	Type of script being executed	op
<junos-context> <user-context>	<class-name>	Login class of the user executing the script	superuser
	<login-name>	Login name of the user executing the script. For AAA access, this is the RADIUS/TACACS username.	jsmith
	<uid>	User ID number of the user executing the script as defined in the device configuration	2999
	<user>	Local name of the user executing the script. Junos OS uses the local name for authentication. It might differ from the <b>login-name</b> used for AAA authentication.	root
<junos-context> <op-context>  (op scripts only)	<via-url>	Empty element included if the remote op script is executed using the <b>op url</b> command	

Table 9: Global Variable \$junos-context Available to Automation Scripts (*continued*)

Parent Node	Node	Description	Example content
<junos-context> <commit-context>  (commit scripts only)	<commit-boot/>	Empty element included when the commit occurs at boot time	
	<commit-check/>	Empty element included when a <b>commit check</b> is performed	
	<commit-comment>	User comment regarding the commit	Commit to fix forwarding issue
	<commit-confirm/>	Empty element included when a <b>commit confirmed</b> is performed	
	<commit-sync/>	Empty element included when a <b>commit synchronize</b> is performed	
	<database-path/>	Element specifying the location of the session's pre-inheritance candidate configuration. For normal configuration sessions, the value of the element is the location of the normal candidate database. For private configuration sessions, the value of the element is the location of the private candidate database. When the <b>&lt;get-configuration&gt; database-path</b> attribute is set to this value, the commit script retrieves the corresponding pre-inheritance candidate configuration.	

The **\$junos-context** variable is a node-set. Therefore, you can access the child elements throughout a script by including the proper XPath expression. The following example commit script writes a message to the system log file if the commit is performed during initial boot-up. The message is given a facility value of **daemon** and a severity value of **info**. For more information, see [syslog\(\)](#).

```

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

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

match configuration {
  if ($junos-context/commit-context/commit-boot) {
    expr jcs:syslog("daemon.info", "This is boot-time commit");
  }
  else {
    /* Do this ... */
  }
}
```

#### Related Documentation

- [Junos Script Automation: Understanding Extension Functions in the jcs and slax Namespaces on page 11](#)

- [Junos Script Automation: Named Templates in the jcs Namespace Overview on page 40](#)
- [SLAX Parameters Overview on page 93](#)
- [SLAX Variables Overview on page 97](#)
- [XSLT Parameters Overview on page 70](#)
- [XSLT Variables Overview on page 73](#)

## CHAPTER 4

# XML

- [XML Overview on page 57](#)
- [XML and Junos OS on page 59](#)

## XML Overview

---

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

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

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

The following sections discuss general aspects of XML:

- [Tag Elements on page 57](#)
- [Attributes on page 58](#)
- [Namespaces on page 58](#)
- [Document Type Definition on page 59](#)

## Tag Elements

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

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

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

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

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

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

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

## Attributes

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

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

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

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

## Namespaces

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



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

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

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

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

## Document Type Definition

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

### Related Documentation

- [Junos XML API and Junos XML Management Protocol Overview on page 7](#)
- [XML and Junos OS on page 59](#)

## XML and Junos OS

Extensible Markup Language (XML) is a standard for representing and communicating information. It is a metalanguage for defining customized tags that are applied to a data set or document to describe the function of individual elements and codify the hierarchical relationships between them. Junos OS natively supports XML for the operation and configuration of devices running Junos OS.

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

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

To display operational mode command output as NETCONF and Junos XML tag elements instead of as the default formatted ASCII, issue the command, and pipe the output to the **display xml** command. Infrastructure tag elements in the response belong to the Junos XML management protocol instead of the NETCONF XML management protocol. The tag elements that describe Junos OS configuration or operational data belong to the Junos XML API, which defines the Junos content that can be retrieved and manipulated by both the Junos XML management protocol and the NETCONF XML management protocol operations. The following example compares the text and XML output for the **show chassis alarms** operational mode command:

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

```
user@host> show chassis alarms | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos">
  <alarm-information xmlns="http://xml.juniper.net/junos/10.4R1/junos-alarm">
    <alarm-summary>
      <no-active-alarms/>
    </alarm-summary>
  </alarm-information>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

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

```
user@host> show chassis alarms | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos">
  <rpc>
    <get-alarm-information>
  </get-alarm-information>
  </rpc>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

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

When you issue the **show chassis alarms** operational mode command, the CLI converts the command into its equivalent Junos XML API request tag **<get-alarm-information>** and sends the XML request to the Junos infrastructure for processing. Junos OS processes the request and returns the **<alarm-information>** response tag element to the CLI. The

CLI then converts the XML output into the “No alarms currently active” message that is displayed to the user.

Junos automation scripts use XML to communicate with the host device. Junos OS provides XML-formatted input to a script. The script processes the input source tree and then returns XML-formatted output to Junos OS. The script type determines the XML input document that is sent to the script as well as the output document that is returned to Junos OS for processing. Commit script input consists of an XML representation of the post-inheritance candidate configuration file. Event scripts receive an XML document containing the description of the triggering event. All script input documents contain a common node-set with information pertaining to the Junos OS environment.

- Related Documentation**
- *Junos XML API Configuration Developer Reference*
  - *Junos XML API Operational Developer Reference*



## CHAPTER 5

# XSLT

- [XSLT Overview on page 63](#)
- [XSLT Namespace on page 65](#)
- [XPath Overview on page 65](#)
- [XSLT Templates Overview on page 68](#)
- [XSLT Parameters Overview on page 70](#)
- [XSLT Variables Overview on page 73](#)
- [XSLT Programming Instructions Overview on page 74](#)
- [XSLT Recursion Overview on page 77](#)
- [XSLT Context \(Dot\) Overview on page 78](#)

### XSLT Overview

---

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

- [XSLT Advantages on page 63](#)
- [XSLT Engine on page 64](#)
- [XSLT Concepts on page 64](#)

### XSLT Advantages

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

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

these configuration statements to instruct the Junos OS user-interface (UI) components to perform the desired behavior.

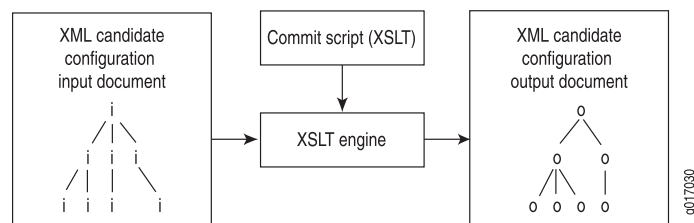
Although XSLT provides a powerful scripting ability, its focus is specific and limited. It does not make Junos OS vulnerable to arbitrary or malicious programmers. XSLT restricts programmers from performing haphazard operations, such as opening random Transmission Control Protocol (TCP) ports, forking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

## XSLT Engine

XSLT is a language for transforming one XML document into another XML document. The basic model is that an XSLT engine (or processor) reads a script (or style sheet) and an XML document. The XSLT engine uses the instructions in the script to process the XML document by traversing the document's hierarchy. The script indicates what portion of the tree should be traversed, how it should be inspected, and what XML should be generated at each point. For commit scripts, op scripts, and event scripts, the XSLT engine is a function of the Junos OS management process (mgd).

Figure 1 on page 64 shows the relationship between an XSLT commit script and the XSLT engine.

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



## XSLT Concepts

XSLT has seven basic concepts. These are summarized in Table 10 on page 64.

**Table 10: XSLT Concepts**

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

Table 10: XSLT Concepts (*continued*)

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

#### Related Documentation

- [XPath Overview on page 65](#)
- [XSLT Context \(Dot\) Overview on page 78](#)
- [XSLT Parameters Overview on page 70](#)
- [XSLT Programming Instructions Overview on page 74](#)
- [XSLT Recursion Overview on page 77](#)
- [XSLT Templates Overview on page 68](#)
- [XSLT Variables Overview on page 73](#)

## XSLT Namespace

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

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

Once the XSLT namespace is declared in a script, you use elements and attributes from the namespace by adding the associated prefix, which in this case is **xsl**, to the tag or attribute name. In the preceding example, the XSLT processor knows to treat **xsl:template** as an XSLT instruction. During processing, the **xsl** prefix is expanded into the URI reference, and the functionality of the **template** element is defined by the XSLT namespace. For more information about namespaces, see [“XML Overview” on page 57](#).

## XPath Overview

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

## Nodes and Axes

XPath views every piece of the document hierarchy as a *node*. For commit scripts, op scripts, and event scripts, the important types of nodes are *element nodes*, *text nodes*, and *attribute nodes*. Consider the following XML tags:

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

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

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

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

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



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

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

---

## Path and Predicate Syntax

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



- **accounting-options**—Selects an element node named **accounting-options** that is a child of the current context.
- **server/name**—Selects an element node named **name** that is a child of an element named **server** that is a child of the current context.
- **/configuration/system/domain-name**—Selects an element node named **domain-name** that is the child of an element named **system** that is the child of the root element of the document (**configuration**).
- **parent::system/host-name**—Selects an element node named **host-name** that is the child of an element named **system** that is the parent of the current context node. The **parent::** axis can be abbreviated as two periods (**..**).

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

- **server[name = '10.1.1.1']**—Selects an element named **server** that is a child of the current context and has a child element named **name** whose value is 10.1.1.1.
- **\*[@inactive]**—Selects any node (**\*** matches any node) that is a child of the current context and that has an attribute (**@** selects nodes from the **attribute** axis) named **inactive**.
- **route[starts-with(next-hop, '10.10.')]** —Selects an element named **route** that is a child of the current context and that has a child element named **next-hop** whose value starts with the string 10.10..

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

## XPath Operators

XPath supports standard logical operators, such as **AND** and **|** (or); comparison operators, such as **=**, **!=**, **<**, and **>**; and numerical operators, such as **+**, **-**, and **\***.

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

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

## XSLT Templates Overview

---

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

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

- [Unnamed \(Match\) Templates on page 68](#)
- [Named Templates on page 69](#)

### Unnamed (Match) Templates

Unnamed templates, also known as match templates, include a **match** attribute that contains an XPath expression to specify the criteria for nodes upon which the template should be invoked. In the following example, the template applies to the element named **route** that is a child of the current context and that has a child element named **next-hop** whose value starts with the string **10.10..**

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

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

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

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

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

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

For more information about how unnamed templates are used in scripts, see [xsl:template match="/" Template](#).

## Named Templates

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

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

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

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

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

When you invoke a named template, you can pass arguments into the template by including the `<xsl:with-param>` child element and specifying the **name** attribute. The value of the `<xsl:with-param>` **name** attribute must match a parameter defined in the actual template; otherwise the parameter is ignored. Optionally, you can set a value for each parameter with either the **select** attribute or the content of the `<xsl:with-param>` element. If you do not define a value for the parameter in the calling environment, the script passes in the current value of the parameter if it was previously initialized, or it generates an error if the parameter was never declared. For more information about passing parameters, see [“XSLT Parameters Overview” on page 70](#).

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

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

For an example showing how to use named templates in a commit script, see *Example: Requiring and Restricting Configuration Statements*.

- Related Documentation**
- [XSLT Parameters Overview on page 70](#)
  - [xsl:apply-templates on page 248](#)
  - [xsl:call-template on page 248](#)
  - [xsl:param on page 254](#)
  - [xsl:template on page 256](#)
  - [xsl:template match="/" Template on page 257](#)
  - [xsl:with-param on page 261](#)

---

## XSLT Parameters Overview

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

### Declaring Parameters

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

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

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

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

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

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

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

## Passing Parameters

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

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

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

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

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

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

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

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

## Example: Parameters and Match Templates

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

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

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

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

## Example: Parameters and Named Templates

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

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

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

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

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

```
<xsl:template match="system">
  <xsl:call-template name="report-changed">
```

```

        <xsl:with-param name="dot" select="..."/>
    </xsl:call-template>
</xsl:template>

```

- Related Documentation**
- [XSLT Templates Overview on page 68](#)
  - [xsl:param on page 254](#)
  - [xsl:with-param on page 261](#)

## XSLT Variables Overview

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

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

```

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

```

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

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

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

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

```

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

```

```

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

```

Table 11 on page 74 provides examples of XSLT variable declarations along with pseudocode explanations.

**Table 11: Examples and Pseudocode for XSLT Variable Declaration**

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

Related Documentation

- [xsl:variable on page 260](#)

## XSLT Programming Instructions Overview

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

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

- [<xsl:choose> Programming Instruction on page 74](#)
- [<xsl:for-each> Programming Instruction on page 75](#)
- [<xsl:if> Programming Instruction on page 75](#)
- [Sample XSLT Programming Instructions and Pseudocode on page 76](#)

### <xsl:choose> Programming Instruction

The `<xsl:choose>` instruction is a conditional construct that causes different instructions to be processed in different circumstances. It is similar to a switch statement in traditional programming languages. The `<xsl:choose>` instruction contains one or more `<xsl:when>` elements, each of which tests an XPath expression. If the test evaluates to true, the XSLT processor executes the instructions in the `<xsl:when>` element. After the XSLT processor finds an XPath expression in an `<xsl:when>` element that evaluates to true, the XSLT processor ignores all subsequent `<xsl:when>` elements contained in the `<xsl:choose>` instruction, even if their XPath expressions evaluate to true. In other words, the XSLT processor processes only the instructions contained in the first `<xsl:when>` element



whose **test** attribute evaluates to true. If none of the `<xsl:when>` elements' **test** attributes evaluate to true, the content of the optional `<xsl:otherwise>` element, if one is present, is processed.

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

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

## <xsl:for-each> Programming Instruction

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

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

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

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

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

## <xsl:if> Programming Instruction

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

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

There is no corresponding else clause.

## Sample XSLT Programming Instructions and Pseudocode

Table 12 on page 76 presents examples that use several XSLT programming instructions along with pseudocode explanations.

**Table 12: Examples and Pseudocode for XSLT Programming Instructions**

Programming Instruction	Pseudocode Explanation
<pre>&lt;xsl:choose&gt; &lt;xsl:when test="system/host-name"&gt;   &lt;change&gt;     &lt;system&gt;       &lt;host-name&gt;M320&lt;/host-name&gt;     &lt;/system&gt;   &lt;/change&gt; &lt;/xsl:when&gt; &lt;xsl:otherwise&gt;   &lt;xnm:error&gt;     &lt;message&gt;       Missing [edit system host-name] M320.     &lt;/message&gt;   &lt;/xnm:error&gt; &lt;/xsl:otherwise&gt; &lt;/xsl:choose&gt;</pre>	<p>When the <b>host-name</b> statement is included at the <b>[edit system]</b> hierarchy level, change the hostname to <b>M320</b>.</p> <p>Otherwise, issue the warning message: <b>Missing [edit system host-name] M320</b>.</p>
<pre>&lt;xsl:for-each select="interfaces/interface[starts-with(name, 'ge-')]/unit"&gt;</pre>	<p>For each Gigabit Ethernet interface configured at the <b>[edit interfaces ge-fpc/pic/port unit logical-unit-number]</b> hierarchy level.</p>
<pre>&lt;xsl:for-each select="data[not(value)]/name"&gt;</pre>	<p>Select any macro parameter that does not contain a parameter value.</p> <p>In other words, match all <b>apply-macro</b> statements of the following form:</p> <pre>apply-macro apply-macro-name {   parameter-name; }</pre> <p>And ignore all <b>apply-macro</b> statements of the form:</p> <pre>apply-macro apply-macro-name {   parameter-name parameter-value; }</pre>
<pre>&lt;xsl:if test="not(system/host-name)"&gt;</pre>	<p>If the <b>host-name</b> statement is not included at the <b>[edit system]</b> hierarchy level.</p>
<pre>&lt;xsl:if test="apply-macro[name = 'no-igp']"</pre>	<p>If the <b>apply-macro</b> statement named <b>no-igp</b> is included at the current hierarchy level.</p>
<pre>&lt;xsl:if test="not(..//apply-macro[name = 'no-ldp'])"</pre>	<p>If the <b>apply-macro</b> statement with the name <b>no-ldp</b> is not included two hierarchy levels above the current hierarchy level.</p>

Related • [xsl:choose on page 249](#)  
Documentation

- [xsl:for-each on page 251](#)
- [xsl:if on page 252](#)
- [xsl:otherwise on page 253](#)
- [xsl:when on page 261](#)

## XSLT Recursion Overview

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

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

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

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

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

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

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

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

## XSLT Context (Dot) Overview

---

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

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

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

## CHAPTER 6

# SLAX

- [SLAX Overview on page 79](#)
- [Converting Scripts Between SLAX and XSLT on page 81](#)
- [SLAX Syntax Rules Overview on page 83](#)
- [SLAX Elements and Element Attributes Overview on page 85](#)
- [XPath Expressions Overview for SLAX on page 87](#)
- [SLAX Templates Overview on page 88](#)
- [SLAX Functions Overview on page 91](#)
- [SLAX Parameters Overview on page 93](#)
- [SLAX Variables Overview on page 97](#)
- [SLAX Statements Overview on page 100](#)
- [XSLT Elements Without SLAX Equivalents on page 104](#)
- [SLAX Operators on page 104](#)

## SLAX Overview

---

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

- [SLAX Advantages on page 79](#)
- [How SLAX Works on page 80](#)

## SLAX Advantages

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

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

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

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

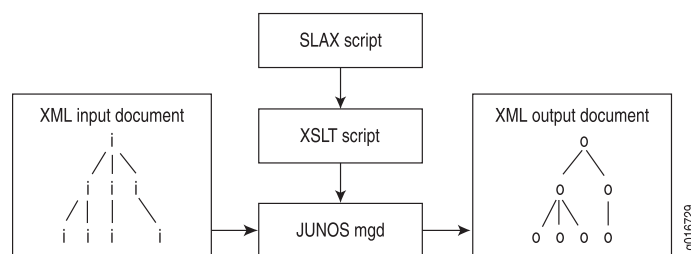
## How SLAX Works

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

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

Figure 2 on page 80 shows the flow of SLAX script input and output.

**Figure 2: SLAX Script Input and Output**



### Related Documentation

- [Converting Scripts Between SLAX and XSLT on page 81](#)
- [SLAX Elements and Element Attributes Overview on page 85](#)

- [SLAX Statements Overview on page 100](#)
- [SLAX Syntax Rules Overview on page 83](#)
- [SLAX Templates Overview on page 88](#)
- [SLAX Variables Overview on page 97](#)
- [XPath Expressions Overview for SLAX on page 87](#)
- [XSLT Overview on page 63](#)

---

## Converting Scripts Between SLAX and XSLT

SLAX is a C-like alternative syntax to XSLT and can be viewed as a preprocessor for XSLT. Before Junos OS invokes the XSLT processor, the software converts any SLAX constructs in the script (such as **if/else if/else**) to equivalent XSLT constructs (such as **<xsl:choose>** and **<xsl:if>**). For more information about SLAX, see [“SLAX Overview” on page 79](#).

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

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

- [Converting a Script from SLAX to XSLT on page 81](#)
- [Converting a Script from XSLT to SLAX on page 82](#)

### Converting a Script from SLAX to XSLT

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

The command syntax is:

```
user@host> request system scripts convert slax-to-xslt source source/filename destination  
destination/<filename> <partial>
```

Starting with Junos OS Release 12.2, you can include the **partial** option to convert partial script input.

The following three examples convert a script from SLAX to XSLT using a source and destination directory relevant to the default storage location for the type of script being converted:

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

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

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

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

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

If you specify only the destination directory and do not specify a destination filename, the generated filename is **SLAX-Conversion-Temp** or **slax-temp** depending on the Junos OS release, with a randomly generated, five-character, alpha-numeric extension.

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

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

## Converting a Script from XSLT to SLAX

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

The command syntax is:

```
user@host> request system scripts convert xslt-to-slax source source/filename destination
destination/<filename> <partial> <version (1.0 | 1.1)>
```

To convert partial script input, include the **partial** option in the command. The **version** option specifies the SLAX version that will be listed in the version statement of the generated script. Specify the version as either 1.0 or 1.1. The default is 1.1. The **partial** and **version** options are supported starting with Junos OS Release 12.2.

The following three examples convert a script from XSLT to SLAX using a source and destination directory relevant to the default storage location for the type of script being converted:

```
user@host> request system scripts convert xslt-to-slax source /var/db/scripts/op/script1.xsl
destination /var/db/scripts/op/script1.slax version 1.0
conversion complete
```

```
user@host> request system scripts convert xslt-to-slax source /var/db/scripts/event/script1.xsl
destination /var/db/scripts/event/script1.slax version 1.0
```



```
conversion complete
```

```
user@host> request system scripts convert xslt-to-slax source /var/db/scripts/commit/script1.xml
destination /var/db/scripts/commit/script1.slax version 1.0
conversion complete
```

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

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

The SLAX script boilerplate lists the specified SLAX version. In this example, the version is 1.0.

```
user@host> file show /var/db/scripts/op/script1.slax
/* Machine Crafted with Care (tm) by slaxWriter */
version 1.0;
...
```

If you specify only the destination directory and do not specify a destination filename, the generated filename is **SLAX-Conversion-Temp** or **slax-temp** depending on the Junos OS release, with a randomly generated, five-character, alpha-numeric extension.

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

```
user@host> file list /var/db/scripts/op
slax-temp.Vosnd
script1.xml
```

**Related Documentation**

- [SLAX Overview on page 79](#)

## SLAX Syntax Rules Overview

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

- [Code Blocks on page 83](#)
- [Comments on page 84](#)
- [Line Termination on page 84](#)
- [Strings on page 85](#)

### Code Blocks

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

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

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

## Comments

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

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

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

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

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

The XSLT equivalent is:

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

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

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

## Line Termination

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

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

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

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

## Strings

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

For example:

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

### Related Documentation

- [SLAX Elements and Element Attributes Overview on page 85](#)
- [SLAX Overview on page 79](#)
- [SLAX Statements Overview on page 100](#)
- [SLAX Templates Overview on page 88](#)
- [SLAX Variables Overview on page 97](#)

## SLAX Elements and Element Attributes Overview

### SLAX Elements

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

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

```
<five> {  
  <six>;  
}  
}
```

The XSLT equivalent is:

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

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

## SLAX Element Attributes

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

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

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

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

The XSLT equivalent is:

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

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

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

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

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

The XSLT equivalent is:

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

**Related Documentation** • [XML Overview on page 57](#)

## XPath Expressions Overview for SLAX

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

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

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

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

The XSLT equivalent is:

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

- Related Documentation**
- [concat\(\) on page 241](#)
  - [SLAX Elements and Element Attributes Overview on page 85](#)
  - [SLAX Syntax Rules Overview on page 83](#)
  - [XPath Overview on page 65](#)
  - [xsl:text on page 259](#)
  - [xsl:value-of on page 259](#)

---

## SLAX Templates Overview

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

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

- [Unnamed \(Match\) Templates on page 88](#)
- [Named Templates on page 89](#)

### Unnamed (Match) Templates

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

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

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

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

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

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

The XSLT equivalent:

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

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

## Named Templates

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

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

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

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

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

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

In SLAX, you invoke named templates using the **call** statement, which consists of the **call** keyword and template name followed by a set of parameter bindings. These bindings are a comma-separated list of parameter names that are passed into the template from the calling environment. Parameter assignments are made by name and not by position in the list. Alternatively, you can declare parameters inside the **call** block using the **with** statement. Parameters passed into a template must match a parameter defined in the actual template; otherwise the parameter is ignored. Optionally, you can set a value for each parameter. If you do not define a value for the parameter in the calling environment, the script passes in the current value of the parameter if it was previously initialized, or

it generates an error if the parameter was never declared. For more information about passing parameters, see [“SLAX Parameters Overview” on page 93](#).

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

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

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

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

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

The XSLT equivalent is:

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

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



```
</output>
</xsl:template>
```

#### Related Documentation

- [SLAX Parameters Overview on page 93](#)
- [XSLT Templates Overview on page 68](#)
- [apply-templates on page 195](#)
- [call on page 199](#)
- [match on page 215](#)
- [param on page 225](#)
- [with on page 239](#)

## SLAX Functions Overview

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

Functions have several advantages over templates, including the following:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Related Documentation
- [function on page 210](#)
  - [param on page 225](#)
  - [result on page 229](#)

## SLAX Parameters Overview

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

- [Declaring Parameters on page 93](#)
- [Passing Parameters to Templates on page 94](#)
- [Example: Parameters and Match Templates on page 95](#)
- [Passing Parameters to Functions on page 96](#)

## Declaring Parameters

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

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

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

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

/* global parameter */
param $interface1 = "fxp0";
```

In a template, you declare parameters either in a parameter list or by using the **param** statement in the template block. Optionally, declare default values for each template parameter. If a template is invoked without the parameter, the default expression is evaluated, and the results are assigned to the parameter. If you do not define a default value in the template, the parameter defaults to an empty string.

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

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

An alternative, but equivalent, declaration is:

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

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

In a function, you declare parameters either in a parameter list or by using the **param** statement in the function block. Optionally, you can declare default values for trailing parameters. If you invoke a function without that trailing parameter, the default expression is evaluated, and the results are assigned to the parameter. If you do not define a default value, the parameter defaults to an empty string.

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

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

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

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

## Passing Parameters to Templates

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

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

```

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

```

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

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

```

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

```

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

```

match configuration {
  call print-host-name($message);
  /* If $message was initialized previously, the current value is used;
   * If $message was declared but not initialized, an empty string is used;
   * If $message was never declared, the call generates an error. */
}

```

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

```

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

```

## Example: Parameters and Match Templates

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

```

match configuration {

```

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

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

The XSLT equivalent is:

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

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

## Passing Parameters to Functions

Version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases, supports functions. Although you can use the **param** statement to define function parameters, you cannot use the **with** statement to pass parameter values into the function from the calling environment. When you call a function, you pass arguments into the function in a comma-separated list. Function arguments are passed to the function by position rather than by name as in a template.

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

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

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

function my:size () {
  param $width;
  param $height;
```

```

    param $scale = 1;
    result <size> {
        expr $width * $height * $scale;
    }
}

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

```

#### Related Documentation

- [SLAX Functions Overview on page 91](#)
- [SLAX Templates Overview on page 88](#)
- [function on page 210](#)
- [param on page 225](#)
- [template on page 232](#)
- [with on page 239](#)

## SLAX Variables Overview

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

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

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

- [Immutable variables on page 97](#)
- [Mutable variables on page 98](#)

### Immutable variables

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

In the declaration, the variable name is prefixed with the dollar sign (\$), which is unlike the XSLT declaration, where the dollar sign does not prefix the value of the **name** attribute

of the `<xsl:variable>` element. Once you declare a variable, you can reference it within an XPath expression using the variable name prefixed with a dollar sign (\$). You initialize a variable by following the variable name with an equal sign (=) and an expression.

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

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

The XSLT equivalent is:

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

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

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

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

```
1, 1  
1, 2  
1, 3  
1, 4  
1, 5
```

## Mutable variables

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

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

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

```
mvar $location = $dot/@location;
```



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

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

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

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

```
version 1.1;

var $list := {
  <list> {
    <grocery> {
      <name> "milk";
      <type> "nonfat";
      <brand> "any";
      <size> "gallon";
    }
    <grocery> {
      <name> "orange juice";
      <type> "no pulp";
      <brand> "any";
      <size> "half gallon";
    }
    <drugstore> {
      <name> "aspirin";
      <brand> "any";
      <size> "50 tablets";
    }
  }
}

match / {

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

```
}  
}
```

The output from the script is:

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

**Related  
Documentation**

- [XSLT Variables Overview on page 73](#)
- [SLAX Parameters Overview on page 93](#)
- [append on page 194](#)
- [mvar on page 217](#)
- [set on page 230](#)
- [var on page 237](#)

---

## SLAX Statements Overview

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

- [for-each Statement on page 101](#)
- [if, else if, and else Statements on page 101](#)
- [match Statement on page 102](#)
- [ns Statement on page 103](#)
- [version Statement on page 103](#)

## for-each Statement

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

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

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

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

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

The XSLT equivalent is:

```
<xsl:for-each select="$inventory/chassis/chassis-module/
  chassis-sub-module[part-number = '750-000610']">
  <message>
    <xsl:value-of select="concat('Down rev PIC in ', ../name, ', ', name, ': ',
      description)"/>
  </message>
</xsl:for-each>
```

## if, else if, and else Statements

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

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

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

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

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

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

The XSLT equivalent is:

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

## match Statement

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

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

The XSLT equivalent is:

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

For more information about constructing match templates, see [“SLAX Templates Overview” on page 88](#).

## ns Statement

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

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

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

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

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

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

The XSLT equivalent is:

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

## version Statement

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

```
version 1.0;
```

The XSLT equivalent is:

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

### Related Documentation

- [else on page 203](#)
- [for-each on page 208](#)
- [if on page 211](#)
- [match on page 215](#)

- [version on page 237](#)

## XSLT Elements Without SLAX Equivalents

---

Some XSLT elements are not directly translated into SLAX statements. Some examples of XSLT elements for which there are no SLAX equivalents in SLAX version 1.0 are `<xsl:fallback>`, `<xsl:output>`, and `<xsl:sort>`.

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

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

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

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

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

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

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

## SLAX Operators

---

SLAX provides a variety of operators, which add great versatility to the SLAX scripting language. [Table 13 on page 105](#) summarizes the available operators and provides an example and an explanation of each.

Table 13: SLAX Operators

Name	Operator	Example / Explanation
Addition	+	<pre>var \$result = 1 + 1;</pre> <p>Return the sum of the operands. This example assigns a value of 2 to the <b>\$result</b> variable.</p>
And	&&	<pre>(\$byte-count &gt; 500000) &amp;&amp; (\$byte-count &lt; 1000000)</pre> <p>Evaluate two expressions and return one boolean result. If either of the two expressions evaluates to <b>false</b>, then the combined expression evaluates to <b>false</b>.</p>
Assignment	=	<pre>var \$mtu = 1500; mvar \$mtu2 = 48; set \$mtu2 = 1500;</pre> <p>Assign a value to a variable or parameter or assign a namespace to a prefix. The example assigns a value of 1500 to both the <b>\$mtu</b> variable and the <b>\$mtu2</b> mutable variable. <b>\$mtu2</b> was originally initialized with a value of 48.</p>
Conditional	?:	<pre>var \$result = (\$a &lt; 10) ? \$b : \$c;</pre> <p>Provide conditional assignment based on the boolean value of the evaluated condition. If the conditional expression evaluates to <b>true</b>, the entire expression assumes the value of the operand to the left of the colon. If the conditional expression evaluates to <b>false</b>, the entire expression assumes the value of the operand to the right of the colon. This operator was introduced in version 1.1 of the SLAX language, which is supported starting with Junos OS Release 12.2.</p> <p>In the example, if the value stored in the variable <b>\$a</b> is less than 10, <b>\$result</b> is assigned the value stored in <b>\$b</b>. Otherwise, <b>\$result</b> is assigned the value stored in <b>\$c</b>.</p>
Division	div	<pre>&lt;output&gt;\$bit-count div 8;</pre> <p>Return the result of dividing the left operand by the right operand. If the remainder of the division is nonzero, the result is expressed in decimal floating-point notation. The example divides the <b>\$bit-count</b> variable by eight, returning the byte count (requires that <b>\$bit-count</b> has been initialized).</p>
Equality	==	<pre>\$mtu == 1500</pre> <p>Return <b>true</b> if the values of the left and right operands are equal; otherwise, the expression returns <b>false</b>. In the example, if <b>\$mtu</b> equals 1500, then the expression resolves to <b>true</b>; otherwise, it returns false (requires that <b>\$mtu</b> has been initialized).</p>
Greater than	>	<pre>\$hop-count &gt; 0</pre> <p>Return <b>true</b> if the value of the left operand is greater than the value of the right operand; otherwise, the expression returns <b>false</b>. In this example, if <b>\$hop-count</b> is greater than zero, the expression returns <b>true</b> (requires that <b>\$hop-count</b> has been initialized).</p>
Greater than or equal to	>=	<pre>\$hop-count &gt;= 1</pre> <p>Return <b>true</b> if the value of the left operand is either greater than or equal to the value of the right operand; otherwise, the expression returns <b>false</b>. In this example, if <b>\$hop-count</b> is 1 or greater, the expression returns <b>true</b> (requires that <b>\$hop-count</b> has been initialized).</p>

Table 13: SLAX Operators (*continued*)

Name	Operator	Example / Explanation
Inequality	<code>!=</code>	<p><code>\$mtu != 1500</code></p> <p>Return <b>true</b> if the values of the left and right operands are not equal; otherwise, the expression returns <b>false</b>. In the example, if <code>\$mtu</code> does not equal 1500, then the expression resolves to <b>true</b>; otherwise, the expression returns <b>false</b> (requires that <code>\$mtu</code> has been initialized)</p>
Iteration	<code>...</code>	<p><code>for \$i (1 ... 10) {   &lt;player number=\$i&gt;; }</code></p> <p>Iterate through a range of integer values with a start value equal to the left operand and an end value equal to the right operand. If the left operand is greater than the right, the numbers are generated in decreasing order. The operator translates into an XPath function that generates the sequence as a node set. This operator was introduced in version 1.1 of the SLAX language, which is supported starting with Junos OS Release 12.2.</p>
Less than	<code>&lt;</code>	<p><code>\$hop-count &lt; 15</code></p> <p>Return <b>true</b> if the value of the left operand is less than the value of the right operand; otherwise, the expression returns <b>false</b>. In this example, if <code>\$hop-count</code> is less than 15, the expression returns <b>true</b> (requires that <code>\$hop-count</code> has been initialized).</p>
Less than or equal to	<code>&lt;=</code>	<p><code>\$hop-count &lt;= 14</code></p> <p>Return <b>true</b> if the value of the left operand is either less than or equal to the value of the right operand; otherwise, the expression returns <b>false</b>. In this example, if <code>\$hop-count</code> is 14 or less, the expression returns <b>true</b> (requires that <code>\$hop-count</code> has been initialized).</p>
Modulo	<code>mod</code>	<p><code>&lt;output&gt; 10 mod 3;</code></p> <p>Return the division remainder of two numbers. In this example, the expression outputs a value of 1.</p>
Multiplication	<code>*</code>	<p><code>&lt;output&gt; 5 * 10;</code></p> <p>Return the product of the operands. In this example, the expression outputs a value of 50.</p>
Node Set, append to	<code>+=</code>	<p><code>mvar \$block = &lt;block&gt; "start here"; append \$block += &lt;block&gt; "next block";</code></p> <p>Append a value to a node set contained in a mutable variable, which is defined with the <b>mvar</b> statement. This operator was introduced in version 1.1 of the SLAX language, which is supported starting with Junos OS Release 12.2.</p>
Node Set Conversion	<code>:=</code>	<p><code>var \$new-node-set := \$rtf-variable;</code></p> <p>Convert a result tree fragment into a node set. A result tree fragment contains an unparsed XML data structure. It is not possible to retrieve any of the embedded XML information from this data type. A script can convert the result tree fragment into a node set and then search the node set for the appropriate information and extract it. This operator is supported in Junos OS Release 9.2 and later releases.</p>



Table 13: SLAX Operators (*continued*)

Name	Operator	Example / Explanation
Or		<p><code>(\$mtu-size != 1500)    (\$mtu-size &gt; 2000)</code></p> <p>Evaluate two expressions and return one boolean result. If either of the two expressions evaluates to true, then the combined expression evaluates to true.</p>
Parentheses	( )	<p><code>var \$result = ( \$byte-count * 8 ) + 150;</code></p> <p>Create complex expressions. Parentheses function the same way as in a mathematical expression, where the expression within the parentheses is evaluated first. Parentheses can be nested; the innermost set of parentheses is evaluated first, then the next set, and so on.</p>
String concatenation	_ (underscore)	<p><code>var \$combined-string = \$host-name _ " is located at " _ \$location;</code></p> <p>Concatenate multiple strings (note that strings cannot be combined using the + operator in SLAX). In the example, if <b>\$host-name</b> is "r1" and <b>\$location</b> is "HQ", then the value of <b>\$combined-string</b> is "r1 is located at HQ".</p>
Subtraction	-	<p><code>var \$result = 64 - 14;</code></p> <p>Return the difference between the left operand and the right operand. This example assigns a value of 50 to the <b>\$result</b> variable.</p>
Unary Minus	-	<p><code>mvar \$number = 5;</code> <code>set \$number = - \$number;</code></p> <p>Negate the value of the operand, changing a positive value to a negative value or a negative value to a positive value. The example negates the value stored in <b>\$number</b> and reassigns the new value of -5 to the variable.</p>
Union		<p><code>var \$all-interface-nodes = \$fe-interface-nodes   \$ge-interface-nodes;</code></p> <p>Create a union of two node sets. All the nodes from one set combine with the nodes in the second set. This is useful when a script needs to perform a similar operation over XML nodes that are pulled from multiple sources.</p>

**Related Documentation**

- [SLAX Elements and Element Attributes Overview on page 85](#)
- [SLAX Overview on page 79](#)
- [SLAX Statements Overview on page 100](#)
- [SLAX Syntax Rules Overview on page 83](#)
- [SLAX Variables Overview on page 97](#)



## CHAPTER 7

# libslax

- [libslax Distribution Overview on page 109](#)
- [libslax Library and Extension Libraries Overview on page 110](#)
- [Understanding the SLAX Processor \(slaxproc\) on page 111](#)

### libslax Distribution Overview

---

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

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

The libslax tools are included as part of the standard Junos OS. However, you can download and install the libslax distribution on a computer with a UNIX-like operating system to develop SLAX scripts outside of Junos OS. Current releases, source code, additional documentation, and support materials for libslax are available at:

<https://github.com/Juniper/libslax>

The SLAX community and support site is available at:

<https://github.com/Juniper/libslax/wiki>

#### Related Documentation

- [Downloading and Installing the libslax Distribution on page 171](#)
- [SLAX Overview on page 79](#)
- [Understanding the SLAX Processor \(slaxproc\) on page 111](#)
- [libslax Library and Extension Libraries Overview on page 110](#)
- [Using the SLAX Processor \(slaxproc\) on page 187](#)

## libslax Library and Extension Libraries Overview

---

- [libslax Library on page 110](#)
- [libslax Extension Libraries on page 110](#)

### libslax Library

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

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

### libslax Extension Libraries

libslax supports a way to dynamically load extension libraries. The libslax distribution includes the xutil, bit, and cURL extension libraries. The source files for the default extension libraries are stored in the **libslax-release/extensions** directory of the distribution. You can supply additional extension libraries beyond the default extension libraries supported by the libslax distribution. Extension library locations can be specified statically at build time or dynamically at run time.

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

There are several ways to specify extension library locations at build time. During installation, to specify a directory prefix other than **/usr/local** for all installation files, including the libraries, execute the **./configure** command and include the **--prefix=prefix** option specifying the location to install the files. The default extension libraries are installed in the **prefix/lib/slax/extensions** directory, and the SLAX processor checks this directory for extension libraries when executing a script. To install just the extension library files in a different, user-defined location, execute the **./configure** command and include the **--with-extensions-dir=dir** option specifying the location where the extension libraries live. The SLAX processor will then automatically check the specified directory for extension libraries when executing a script. For more information about installing libslax, see ["Downloading and Installing the libslax Distribution" on page 171](#).

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

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

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

**Related  
Documentation**

- [libslax bit Extension Library on page 172](#)
- [libslax cURL Extension Library on page 174](#)
- [Downloading and Installing the libslax Distribution on page 171](#)
- [libslax Distribution Overview on page 109](#)

---

## Understanding the SLAX Processor (`slaxproc`)

- [slaxproc Overview on page 111](#)
- [slaxproc Modes on page 112](#)
- [slaxproc Options on page 112](#)
- [slaxproc File Argument Handling on page 114](#)
- [slaxproc UNIX Scripting Support on page 114](#)

### slaxproc Overview

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

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

The syntax for the `slaxproc` command is:

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

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

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

```
$ slaxproc -h
```

## slaxproc Modes

The slaxproc mode defines what function the processor performs. Valid modes include: **check**, which validates SLAX script syntax and content; **format**, which formats a script to correct the indentation and spacing to the preferred style; **run**, which executes a SLAX script; and **convert**, which converts a script between SLAX and XSLT formats.

[Table 14 on page 112](#) outlines the slaxproc modes. The default mode is **--run** or **-r**. If you do not explicitly specify a mode, the SLAX processor executes a script.

**Table 14: slaxproc Modes**

Mode	Description
--check -c	Perform a syntax and content check on a SLAX script, reporting any errors. This mode is useful for off-box syntax checks before installing or uploading scripts to a device running Junos OS.
--format -F	Format a SLAX script, correcting indentation and spacing to the preferred style.
--run -r	Run a SLAX script. This is the default mode. The script name, input file name, and output file name can be provided using command-line options, positional arguments, or a mix of both. Input defaults to standard input, and output defaults to standard output.
--slax-to-xslt -x	Convert a SLAX script into XSLT format. The script filename and output filename are provided using command-line options, positional arguments, or a mix of both.
--xslt-to-slax -s	Convert an XSLT script into SLAX format. The script filename and output filename are provided using command-line options, positional arguments, or a mix of both.

## slaxproc Options

The slaxproc options include file options and common options. File options are used specify the script file, input file, output file, and trace file. Common options include additional functionality and options provided by the SLAX processor such as verbose debugging output. [Table 15 on page 112](#) lists the slaxproc common options and file options.

**Table 15: slaxproc Common Options and File Options**

Option	Description
--debug -d	Enable the SLAX/XSLT debugger.
--empty -E	Provide an empty document as the input data set. This is useful for scripts that do not expect or need meaningful input.

Table 15: slaxproc Common Options and File Options (*continued*)

Option	Description
--exslt -e	Enable the EXSLT library, which provides a set of standard extension functions. See <a href="http://www.exslt.org">http://www.exslt.org</a> for more information.
--help -h	Display the help message and exit.
--html -H	Parse input data using the HTML parser, which differs from XML.
--include <dir> -I <dir>	Add a directory to the list of directories searched when using include and import files. Alternatively, you can define the SLAXPATH environment variable to specify a colon-delimited list of directories to search.
--indent -g	Indent output. This option is identical to the behavior triggered by <b>output-method { indent 'true'; }</b> .
--input <file> -i <file>	Read input from the specified file.
--lib <dir> -L <dir>	Add a directory to the list of directories searched when using extension libraries. Alternatively, you can define the SLAXEXTPATH environment variable to specify a colon-delimited list of extension library locations to search.
--name <file> -n <file>	Read the SLAX script from the specified file.
--no-randomize	Do not initialize the random number generator. This is useful if you want the script to return identical data for a series of invocations. This option is typically only used during testing.
--no-tty	Do not use tty for the SLAX debugger and other tty-related input needs.
--output <file> -o <file>	Write output to the specified file.
--param <name> <value> -a <name> <value>	Pass a parameter to the script using the name and value pair provided. Note that all parameters are string parameters, so normal quoting rules apply.
--partial -p	Allow the input data to contain a partial SLAX script, which can be used with the <b>--slax-to-xslt</b> or <b>-x</b> mode to perform partial transformations.
--trace <file> -t <file>	Write trace data to the specified file.
--verbose -v	Add verbose internal debugging output to the trace data output, including calls to the <b>slaxLog()</b> function.

Table 15: slaxproc Common Options and File Options (*continued*)

Option	Description
<code>--version</code> <code>-V</code>	Show version information and exit.
<code>--write-version &lt;version&gt;</code> <code>-w &lt;version&gt;</code>	Write the specified version number to the output file when converting a script using the <code>--xslt-to-slax</code> or <code>-s</code> mode. This can be used to limit the conversion to avoid using SLAX 1.1 features. Acceptable values are 1.0 and 1.1. If this option is not specified, the SLAX script version defaults to 1.1.

## slaxproc File Argument Handling

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

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

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

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

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

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

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

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

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

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

## slaxproc UNIX Scripting Support

SLAX supports the `"#!"` UNIX scripting mechanism, allowing the first line of a script to begin with the characters `'#'` and `'!'` followed by a path to the executable that runs the script and a set of command-line arguments. For example:

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

or



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

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

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

additional arguments can be provided:

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

and the resulting command becomes:

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

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

#### **Related Documentation**

- [Using the SLAX Processor \(slaxproc\) on page 187](#)
- [libslax Distribution Overview on page 109](#)
- [libslax Library and Extension Libraries Overview on page 110](#)
-



## PART 2

# Configuration

- [Storing and Enabling Scripts on page 119](#)
- [Configuring a Remote Source for Scripts on page 123](#)
- [Configuring Script Synchronization Between Routing Engines on page 135](#)
- [Configuring the Session Protocol for Scripts on page 139](#)
- [Configuring Limits on Event Policies and Memory Allocation for Scripts on page 153](#)
- [Provisioning Services Using Service Template Automation on page 157](#)
- [Using libslax on page 171](#)
- [SLAX Statements on page 193](#)
- [Standard XPath and XSLT Functions Used in Automation Scripts on page 241](#)
- [Standard XSLT Elements and Attributes Used in Automation Scripts on page 247](#)
- [Configuration Statements Common to All Scripts on page 263](#)



## CHAPTER 8

# Storing and Enabling Scripts

- [Storing and Enabling Scripts on page 119](#)
- [Storing Scripts in Flash Memory on page 120](#)
- [Storing and Using Imported Scripts and Script Functionality on page 121](#)

### Storing and Enabling Scripts

---

To use a script on a switch, router, or security device, you must copy the script to the device and enable it in the configuration.

1. Create the script.
2. Copy the script to the appropriate directory on the device for that script type. Only users who belong to the Junos OS **super-user** login class can access and edit files in the script directories on a device running Junos OS.

By default, scripts are stored in and executed from the **/var/db/scripts** directory on the device's hard drive under the subdirectory appropriate to the script type. You can also store scripts on the flash drive in the **/config/scripts** directory under the subdirectory appropriate to the script type.

**commit script**—Copy the script to the **/var/db/scripts/commit** directory on the hard drive or the **/config/scripts/commit** directory on the flash drive.

**op script**—Copy the script to the **/var/db/scripts/op** directory on the hard drive or the **/config/scripts/op** directory on the flash drive.

**event script**—Copy the script to the **/var/db/scripts/event** directory on the hard drive or the **/config/scripts/event** directory on the flash drive.



**NOTE:** If the device has dual Routing Engines and you want to enable the script to execute on both Routing Engines, you must copy it to the appropriate directory on both Routing Engines. The **commit synchronize** command does not automatically copy scripts between Routing Engines.

3. Enable the script by including the **file filename** statement at the appropriate hierarchy level for that script type.

**commit script**—Include the **file filename** statement at the **[edit system scripts commit]** hierarchy level. For instructions, see *Controlling Execution of Commit Scripts During Commit Operations*.

**op script**—Include the **file filename** statement at the **[edit system scripts op]** hierarchy level. For instructions, see *Enabling an Op Script and Defining a Script Alias*.

**event script**—Include the **file filename** statement at the **[edit event-options event-script]** hierarchy level. For instructions, see *Enabling an Event Script*.

4. If you store scripts in and load them from flash memory, include the **load-scripts-from-flash** statement at the **[edit system scripts]** hierarchy level. For detailed information about storing scripts in flash memory, see [“Storing Scripts in Flash Memory” on page 120](#).

```
[edit]
user@host# set system scripts load-scripts-from-flash
```

5. Issue the **commit** command.

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

Newly enabled commit scripts do not execute during the commit operation but execute automatically during each subsequent commit operation. After the commit operation completes, enabled event scripts are loaded into memory and are ready for automatic execution in response to system log events. For more information, see *Executing Event Scripts in an Event Policy*. After the commit operation completes, op scripts can be executed on the device. For more information, see *Executing an Op Script*.

#### Related Documentation

- [Storing Scripts in Flash Memory on page 120](#)
- [Storing and Using Imported Scripts and Script Functionality on page 121](#)
- *Controlling Execution of Commit Scripts During Commit Operations*
- [load-scripts-from-flash \(Scripts\) on page 263](#)

---

## Storing Scripts in Flash Memory

By default, scripts are stored in and executed from the **/var/db/scripts/** directory on the device's hard drive under the subdirectory appropriate to the script type. To store scripts in and load them from flash memory instead, include the **load-scripts-from-flash** statement at the **[edit system scripts]** hierarchy level:

```
[edit]
user@host# set system scripts load-scripts-from-flash
```

When you add the **load-scripts-from-flash** statement in the configuration, all commit, event, operation, and script library scripts are loaded from the **/config/scripts/** directory on the flash drive under the subdirectory appropriate to the script type. You must manually move scripts from the hard drive to the flash drive. They are not moved automatically.

Similarly, if you delete the **load-scripts-from-flash** statement from the configuration, you must manually copy the scripts from the flash drive to the hard drive to ensure that the current versions of the scripts are executed. Changing the scripts' physical location has no effect on their operation.

The **/var/run/scripts/** directory always links to the directory from which the scripts are loaded. If you do not set the **load-scripts-from-flash** statement in the configuration, **/var/run/scripts/** points to the **/var/db/scripts/** directory on the device's hard drive. If you set the **load-scripts-from-flash** statement in the configuration, **/var/run/scripts/** points to the **/config/scripts/** directory in flash memory.

To view the scripts currently on the device, list the contents of **/var/run/scripts/type/**, where *type* is the subdirectory appropriate to the script type. In the following example, the **load-scripts-from-flash** statement is not configured. In this case, **/var/run/scripts/commit/** points to the **/var/db/scripts/commit/** directory on the hard drive. Listing the files for **/var/run/scripts/commit/** is identical to listing the files in the **/var/db/scripts/commit/** directory.

```
user@host> file list /var/run/scripts/commit
```

```
/var/run/scripts/commit:  
commit-changes-load-replace.slax  
commit-protect.slax
```

```
user@host> file list /var/db/scripts/commit
```

```
/var/db/scripts/commit:  
commit-changes-load-replace.slax  
commit-protect.slax
```

```
user@host> file list /config/scripts/commit
```

```
/config/scripts/commit:
```

- Related Documentation**
- [Storing and Enabling Scripts on page 119](#)
  - [Storing and Using Imported Scripts and Script Functionality on page 121](#)
  - [load-scripts-from-flash \(Scripts\) on page 263](#)

---

## Storing and Using Imported Scripts and Script Functionality

Starting with Junos OS Release 11.1, Junos OS provides a dedicated directory for script libraries, where users can store scripts and script functionality that then can be imported into any commit, event, or op script. Upon installation, Junos OS creates the **/var/db/scripts/lib/** directory. Junos OS will not overwrite or erase any files in an existing **lib/** directory upon installation or upgrade.

If you store scripts in and run them from flash memory, both the executed scripts and the imported scripts must be present on the flash drive. When you configure the **load-scripts-from-flash** statement at the **[edit system scripts]** hierarchy level, Junos OS creates the **/config/scripts/lib/** directory. When you add or remove the **load-scripts-from-flash** statement in the configuration, you must manually move scripts

and script libraries from the hard drive to the flash drive, or vice versa, as appropriate. They are not moved automatically.

Imported scripts must be stored in the `/var/db/scripts/lib/` directory on the hard drive, or if the `load-scripts-from-flash` statement is configured, in the `/config/db/scripts/lib/` directory on the flash drive. To import a script from the script library, include the `<xsl:import>` tag in the style sheet declaration of an XSLT script or the `import` statement in a SLAX script and specify the file location. The following sample code imports the `/var/db/scripts/lib/test.xml` file:

<b>XSLT Syntax</b>	<pre>&lt;?xml version="1.0"?&gt; &lt;xsl:stylesheet version="1.0"&gt;   &lt;xsl:import href="../../../lib/test.xml"/&gt;   ... &lt;/xsl:stylesheet&gt;</pre>
--------------------	--

<b>SLAX Syntax</b>	<pre>version 1.0; import "../../../lib/test.xml";</pre>
--------------------	---

- |                              |  |
|------------------------------|--|
| <b>Related Documentation</b> | <ul style="list-style-type: none"><li>• <a href="#">Storing and Enabling Scripts on page 119</a></li><li>• <a href="#">Storing Scripts in Flash Memory on page 120</a></li></ul> |
|------------------------------|--|



## CHAPTER 9

# Configuring a Remote Source for Scripts

- [Overview of Updating Scripts from a Remote Source on page 123](#)
- [Using a Master Source Location for a Script on page 124](#)
- [Using an Alternate Source Location for a Script on page 129](#)

### Overview of Updating Scripts from a Remote Source

---

You can update the scripts on a device running Junos OS by retrieving a copy from a remote machine (which can be another device running Junos OS or a regular networked computer). This eases file management, because you can make changes to the master script in a single location and then update the copy on each device where the script is currently enabled. Each device continues to use its locally stored scripts, only updating a script when you issue the appropriate operational or configuration mode command.

For each script, you can configure the **source** statement and a URL at the hierarchy level where you configured the script to define the remote location that houses the master copy of that script. When you then issue the **set refresh** configuration mode command for a script, the device running Junos OS updates its local copy by retrieving the remote master copy from that URL.

You can also store a copy of a particular script at a remote location other than the master source. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems. To refresh a single script or multiple scripts from the remote location, you issue the **set refresh-from** configuration mode command at the appropriate hierarchy level and specify the URL. You can also refresh a single script from a remote location using the **request system scripts refresh-from** operational mode command.

You can use the **set refresh** and **set refresh-from** commands to update either an individual script or multiple scripts of a given type on the device. When you issue the **set refresh** or **set refresh-from** command, the switch, router, or security device immediately attempts to connect to the appropriate remote source for each script. If successful, the device updates the local script with the remote source. If a problem occurs, a set of error messages is returned.

Issuing the **set refresh** or **set refresh-from** command does not add the **refresh** and **refresh-from** statements to the configuration. Thus, these commands behave like

operational mode commands by executing an operation, instead of adding a statement to the configuration. The **refresh** and **refresh-from** statements are mutually exclusive.

If a device has dual Routing Engines and you want to update the script on both Routing Engines, you must issue the **set refresh** or **set refresh-from** command on each Routing Engine separately. Alternatively, starting with Junos OS Release 13.2, you can refresh the scripts on the requesting Routing Engine and then use either the **request system scripts synchronize** operational mode command to synchronize scripts to the other Routing Engine or the **commit synchronize scripts** configuration mode command to synchronize all scripts to the other Routing Engine when you commit and synchronize the configuration. If you use the **request system scripts refresh-from** operational mode command to refresh a script from a specific URL, include the **sync** option to refresh the script on both Routing Engines.



**CAUTION:** For commit scripts, we recommend that you do not automate the update function by including the refresh statement as a commit script change element. Even though this might seem like a good way to ensure that the most current commit script is always used, we recommend against it for the following reasons:

- Automated update means that the network must be operational for the commit operation to succeed. If the network goes down after you make a configuration error, you cannot recover quickly.
- If multiple commit scripts need to be updated during each commit operation, the network response time can slow down.
- Automated update is always the last action performed during a commit operation. Consequently, the updated commit script executes only during the next commit operation. This is because commit scripts are applied to the candidate configuration before the software copies any persistent changes generated by the scripts to the candidate configuration. In contrast, if you perform the update operation manually, the updated commit script takes effect as expected, that is, immediately after you commit the refresh statement in the configuration.
- If you automate the update operation, the **refresh-from** statement has no effect, because the **refresh-from** URL conflicts with and is overridden by the **source** statement URL. For information about the **refresh-from** statement, see [“Using an Alternate Source Location for a Script” on page 129](#).

- 
- Related Documentation**
- [Using a Master Source Location for a Script on page 124](#)
  - [Using an Alternate Source Location for a Script on page 129](#)

---

## Using a Master Source Location for a Script

- [Configuring and Refreshing from the Master Source for a Script on page 125](#)
- [Example: Configuring and Refreshing from the Master Source for a Script on page 127](#)

## Configuring and Refreshing from the Master Source for a Script

You can store a master copy of each script in a central repository. This eases file management because you can make changes to the master script in one place and then update the copy on each device where the script is currently enabled. This section discusses the following concepts:

- [Configuring the Master Source for a Script on page 125](#)
- [Updating a Script from the Master Source on page 125](#)

---

### Configuring the Master Source for a Script

To specify the location of the master source for a single script, configure the **source** statement and the URL of the master file. Including the **source** statement in the configuration does not affect the local copy of the script until you issue the **set refresh** command. At that point, the device retrieves the master copy from the specified URL and overwrites the local copy.

The hierarchy location for the **source** statement depends on the script type and filename.

```
[edit system scripts commit file filename]  
user@R1# set source url  
  
[edit system scripts op file filename]  
user@R1# set source url  
  
[edit event-options event-script file filename]  
user@R1# set source url
```

Where

- *filename*—Name of the script.
- *url*—URL of the script's master source file. Specify the source as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

---

### Updating a Script from the Master Source

If you configure a master source for one or more scripts on a device, you can refresh the scripts on that device using the **set refresh** configuration mode command. You can update a single script or all scripts of a given script type that have a master source location configured.

The update operation occurs as soon as you issue the **set refresh** command. When you issue the **set refresh** command, the switch, router, or security device immediately attempts to connect to the specified URL and retrieve a copy of the master file. The master copy overwrites the local script stored in the scripts directory on the device. If a master source is not defined for a script, that script is not updated and a warning is issued. For commit scripts, the updated commit script is executed when you next issue the **commit** command.



**NOTE:** Issuing the `set refresh` command does not add the `refresh` statement to the configuration. Thus the command behaves like an operational mode command by executing an operation, instead of adding a statement to the configuration.

The `set refresh` command is unique in the Junos OS CLI in that it behaves like an operational mode command and yet it can be executed from within configuration mode. All other Junos OS CLI operational mode commands can only be executed from command mode. The functionality is provided in this manner as a convenience to users developing commit scripts.

If the device has dual Routing Engines and you want to update a script on both Routing Engines, you must issue the `set refresh` command on each Routing Engine separately. The `commit synchronize` command does not cause the `refresh` statement to update scripts on both Routing Engines. Alternatively, starting with Junos OS Release 13.2, you can refresh the scripts on the requesting Routing Engine and then use either the `request system scripts synchronize` operational mode command to synchronize scripts to the other Routing Engine or the `commit synchronize scripts` configuration mode command to synchronize all scripts to the other Routing Engine when you commit and synchronize the configuration.

To update a single script from its master source, issue the `set refresh` command at the hierarchy level where the script is configured. The hierarchy location depends on the script type and filename as shown in the following examples. The `source` statement specifying the master source location must already be configured.

```
[edit system scripts commit file filename]  
user@R1# set refresh  
  
[edit system scripts op file filename]  
user@R1# set refresh  
  
[edit event-options event-script file filename]  
user@R1# set refresh
```

Where *filename* is the name of the script.

To update all enabled scripts of a given script type from their master source files, issue the `set refresh` command at the hierarchy level for that script type.

```
[edit system scripts commit]  
user@R1# set refresh  
  
[edit system scripts op]  
user@R1# set refresh  
  
[edit event-options event-script]  
user@R1# set refresh
```

## Example: Configuring and Refreshing from the Master Source for a Script

The following example configures a master source file for an op script on a device running Junos OS. The remote source is defined as an HTTP URL. The example uses the master source to update the local copy of the script on the device.

- [Requirements on page 127](#)
- [Overview on page 127](#)
- [Configuration on page 127](#)
- [Verification on page 128](#)

### Requirements

- Routing, switching, or security device running Junos OS.

### Overview

You can store a master copy of each script in a central repository. You can make changes to the master script in one place and then update the local copy of the script on devices where the script is enabled.

This example enables the op script **iso.xml** on a device running Junos OS and then configures a master source location for the script. The remote source for the **iso.xml** file is the HTTP URL `http://my.example.com/pub/scripts/iso.xml`.

Once you configure the master source location, you refresh the local script by issuing the **set refresh** configuration mode command at the hierarchy level where you configured the script. In this example, you would issue the **set refresh** command at the **[edit system scripts op file iso.xml]** hierarchy level.

### Configuration

#### Step-by-Step Procedure

To download, enable, and configure the master source location for the script:

1. Copy the script to the `/var/db/scripts/op/` directory on the device.
2. In configuration mode, configure the **file** statement to enable the **iso.xml** script.
 

```
[edit system scripts op]
user@R1# set file iso.xml
```
3. To configure the master source for the **iso.xml** file, include the **source** statement and source location at the **[edit system scripts op file iso.xml]** hierarchy level.
 

```
[edit system scripts op file iso.xml]
user@R1# set source http://my.example.com/pub/scripts/iso.xml
```
4. Issue the **commit and-quit** command to commit the configuration and exit to operational mode.
 

```
[edit]
user@R1# commit and-quit
```

**Results**

```
system {
  scripts {
    op {
      file iso.xml {
        source http://my.example.com/pub/scripts/iso.xml;
      }
    }
  }
}
```

### *Verifying the Script*

**Purpose** Verify that the script is on the device and enabled in the configuration.

**Action** Issue the **file list** operational mode command to view the files in the specified directory. The **detail** option provides additional information such as permissions, file size, and modified date.

```
user@R1> file list /var/db/scripts/op detail
```

```
/var/db/scripts/op:
total 128
-rw-r--r--  1 root  admin  13897 Feb 10  2011 iso.xml
...
```

Issue the **show configuration system scripts op** operational mode command to list the op scripts currently enabled on the device.

```
user@R1> show configuration system scripts op
file iso.xml
```

### *Refreshing the Script from the Master Source*

**Step-by-Step Procedure** To refresh the local copy of the script from the master source file:

1. In configuration mode, issue the **set refresh** command at the **[edit system scripts op file iso.xml]** hierarchy level.

```
[edit system scripts op file iso.xml]
user@R1# set refresh
```

---

### **Verification**

#### *Verifying the Updated Script*

**Purpose** After refreshing the script, verify that the local copy is updated.

**Action** Issue the **file list** operational mode command with the **detail** option to view the files in the specified directory. Verify that the modified date reflects the refreshed version.

```
user@R1> file list /var/db/scripts/op detail

/var/db/scripts/op:
total 128
-rw-r--r--  1 root  admin  14128 May 26  2011 iso.xsl
...
```

- Related Documentation**
- [Using an Alternate Source Location for a Script on page 129](#)
  - *refresh (Commit Scripts)*
  - *refresh (Op Scripts)*
  - *refresh (Event Scripts)*

## Using an Alternate Source Location for a Script

- [Refreshing a Script from an Alternate Location on page 129](#)
- [Example: Refreshing a Script from an Alternate Source on page 131](#)

### Refreshing a Script from an Alternate Location

In addition to updating a script from the master source defined by the **source** statement, you also can update a script from an alternate location using the **set refresh-from** configuration mode command or the **request system scripts refresh-from** operational mode command. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems.

The update operation occurs as soon as you issue either the **set refresh-from** configuration mode command or the **request system scripts refresh-from** operational mode command. When you issue the command, the switch, router, or security device immediately attempts to connect to the specified URL and retrieve a copy of the file. The copy overwrites the local script stored in the scripts directory on the device. If a copy of the source is not available at the remote URL, that script is not updated and a warning is issued. For commit scripts, the updated commit script is executed when you next issue the **commit** command.

Issuing the **set refresh-from** command does not add the **refresh-from** statement to the configuration. Thus the **set refresh-from** command behaves like an operational mode command by executing an operation, instead of adding a statement to the configuration.

If a device has dual Routing Engines and you want the script to be updated on both Routing Engines, you must issue the **set refresh-from** command on each Routing Engine separately. The **commit synchronize** command does not cause the **refresh-from** statement to update scripts on both Routing Engines. Alternatively, starting with Junos OS Release 13.2, you can refresh the scripts on the requesting Routing Engine and then use either the **request system scripts synchronize** operational mode command to synchronize scripts to the other Routing Engine or the **commit synchronize scripts** configuration mode command to synchronize all scripts to the other Routing Engine when you commit and

synchronize the configuration. In operational mode, you can use the **request system scripts refresh-from** command with the **sync** option to refresh the script on both Routing Engines.

To update a single script from an alternate source, issue the **set refresh-from** command under the hierarchy level where the script is configured, and specify the location of the remote file. The hierarchy location depends on the script type and filename as shown in the following examples:

```
[edit system scripts commit file filename]
user@R1# set refresh-from url

[edit system scripts op file filename]
user@R1# set refresh-from url

[edit event-options event-script file filename]
user@R1# set refresh-from url
```

To update all enabled scripts of a given script type from an alternate source, issue the **set refresh-from** command at the hierarchy level for that script type, and specify the URL of the remote repository that houses the scripts.

```
[edit system scripts commit]
user@R1# set refresh-from url

[edit system scripts op]
user@R1# set refresh-from url

[edit event-options event-script]
user@R1# set refresh-from url
```

In operational mode, to update a single script from an alternate source, issue the **request system scripts refresh-from** command, and specify the script type, filename, and remote URL.

```
user@R1> request system scripts refresh-from (commit | event | op) file filename url
url <sync>
```

Where

- *filename*—Name of the script.
- *url*—URL of the remote script or directory. Specify the source as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

If you request to refresh a script that does not exist at the remote site, the device generates an error message. For example:

```
user@host> request system scripts refresh-from op file nonexistent.slax url
http://host1.juniper.net/nonexistent.slax
refreshing 'nonexistent.slax' from 'http://host1.juniper.net/nonexistent.slax'
fetch-secure: http://host1.juniper.net/nonexistent.slax: Not Found
error: file-fetch failed
error: communication error: rpc failed (file-fetch)
error: error retrieving file http://host1.juniper.net/nonexistent.slax
```



When you issue the **set refresh-from** command, Junos OS creates a folder in the **/var/tmp** directory, which is used for the file transfer. After the transfer and refresh operations are complete, Junos OS deletes the temporary folder.

### Example: Refreshing a Script from an Alternate Source

The following example uses an alternate source location to update the local copy of the script on a device running Junos OS. The remote source is defined as an HTTP URL.

- [Requirements on page 131](#)
- [Overview on page 131](#)
- [Configuration on page 131](#)
- [Verification on page 132](#)

#### Requirements

- Routing, switching, or security device running Junos OS.

#### Overview

You can update a script from a location other than that of the master source. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems. You can refresh a single script or all scripts of a given type from the alternate location.

This example enables the **op** script **iso.xml** on a device running Junos OS and then refreshes the script from a location other than the master source location. The remote source for the **iso.xml** file is the HTTP URL <http://my.example.com/pub/scripts2/iso.xml>.

You refresh the local script by issuing the **set refresh-from** configuration mode command at the hierarchy level where you configured the script. In this example, you would issue the **set refresh-from** command at the **[edit system scripts op file iso.xml]** hierarchy level.

#### Configuration

##### Step-by-Step Procedure

To download and enable the script:

1. Copy the script to the **/var/db/scripts/op/** directory on the device.
2. In configuration mode, configure the **file** statement to enable the **iso.xml** script.

```
[edit system scripts op]
user@R1# set file iso.xml
```

3. Issue the **commit and-quit** command to commit the configuration and exit to operational mode.

```
[edit]
user@R1# commit and-quit
```

##### Results

```
system {
  scripts {
    op {
      file iso.xml;
```

```
    }  
  }  
}
```

### *Verifying the Script*

**Purpose** Verify that the script is on the device and enabled in the configuration.

**Action** Issue the **file list** operational mode command to view the files in the specified directory. The **detail** option provides additional information such as permissions, file size, and modified date.

```
user@R1> file list /var/db/scripts/op detail
```

```
/var/db/scripts/op:  
total 128  
-rw-r--r--  1 root  admin  13897 Feb 10  2011 iso.xsl  
...
```

Issue the **show configuration system scripts op** operational mode command to list the op scripts currently enabled on the device.

```
user@R1> show configuration system scripts op  
file iso.xsl
```

### *Refreshing the Script from the Alternate Location*

**Step-by-Step Procedure** To refresh the local copy of the script from the alternate location:

1. In configuration mode, issue the **set refresh-from** command at the **[edit system scripts op file iso.xsl]** hierarchy level.

```
[edit system scripts op file iso.xsl]  
user@R1# set refresh-from http://my.example.com/pub/scripts2/iso.xsl
```

---

### **Verification**

#### *Verifying the Updated Script*

**Purpose** After refreshing the script, verify that the local copy is updated.

**Action** Issue the **file list** operational mode command with the **detail** option to view the files in the specified directory. Verify that the modified date reflects the refreshed version.

```
user@R1> file list /var/db/scripts/op detail  
  
/var/db/scripts/op:  
total 128  
-rw-r--r--  1 root  admin  14128 May 26  2011 iso.xsl  
...
```

**Related Documentation**

- [Using a Master Source Location for a Script on page 124](#)
- [request system scripts refresh-from on page 279](#)

- *refresh-from (Commit Scripts)*
- *refresh-from (Op Scripts)*
- *refresh-from (Event Scripts)*



## CHAPTER 10

# Configuring Script Synchronization Between Routing Engines

- [Example: Configuring Script Synchronization Between Routing Engines on page 135](#)

## Example: Configuring Script Synchronization Between Routing Engines

---

This example shows how to configure a device with dual Routing Engines running Junos OS to synchronize all commit, event, lib, and op scripts between Routing Engines every time you execute a **commit synchronize** operation.

- [Requirements on page 135](#)
- [Overview on page 135](#)
- [Configuration on page 136](#)
- [Verification on page 137](#)
- [Troubleshooting on page 137](#)

### Requirements

A routing, switching, or security device with dual Routing Engines running Junos OS Release 13.2 or later is required.

### Overview

In this example, you configure a device with dual Routing Engines running Junos OS to synchronize all commit, event, lib, and op scripts from the requesting Routing Engine to the other Routing Engine whenever you execute a **commit synchronize** command to commit and synchronize the configuration. When configured, the device synchronizes all scripts regardless of whether they are enabled in the configuration.

In this example, the **load-scripts-from-flash** statement is not configured on the requesting Routing Engine. Thus, the device synchronizes the scripts that are on the hard disk of the requesting Routing Engine to the hard disk of the responding Routing Engine.



**NOTE:** On the hard disk, scripts are stored under the `/var/db/scripts` directory in the subdirectory appropriate to the script type. In flash memory, scripts are stored under the `/config/scripts` directory in the subdirectory appropriate to the script type. EX Series switches use the default directory `/config/db/scripts`.

---

## Configuration

### CLI Quick Configuration

To quickly configure this example, copy the following command, paste it in a text file, and then copy and paste the command into the CLI at the **[edit]** hierarchy level.

```
set system scripts synchronize
```

---

### Configuring Script Synchronization for Commit Synchronize Operations

---

### Step-by-Step Procedure

To automatically synchronize scripts between Routing Engines during a **commit synchronize** operation:

1. Configure the **synchronize** statement at the **[edit system scripts]** hierarchy level.

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

2. Commit and synchronize the configuration.

```
[edit system scripts]
user@host# commit synchronize
re0:
configuration check succeeds
re1:
commit complete
re0:
commit complete
```

When you issue the first and subsequent **commit synchronize** commands, the device performs a commit check on the requesting Routing Engine, synchronizes all scripts to the other Routing Engine, synchronizes, performs a commit check, and commits the configuration on the responding Routing Engine, and finally commits the configuration on the requesting Routing Engine.

---

## Results

The resulting configuration is:

```
system {
  scripts {
    synchronize;
  }
}
```

## Verification

Confirm that the configuration is working properly and the synchronization is successful.

### Verifying Script Synchronization

---

**Purpose** Verify that the scripts present on the requesting Routing Engine are synchronized to the other Routing Engine.

In this example, the **load-scripts-from-flash** statement is not configured for the requesting Routing Engine. Therefore, the device synchronizes scripts from the **/var/db/scripts** directory on the requesting Routing Engine to the **/var/db/scripts** directory on the responding Routing Engine.

**Action** Use the **file list** operational mode command to view the files in the **/var/db/scripts** directory on each Routing Engine.

1. On the requesting Routing Engine, list the files under the **/var/db/scripts/** directory.

```
user@host> file list /var/db/scripts/* detail
/var/db/scripts/commit:
-rw-r--r--  1 remote wheel      1014 Jul 17 16:18 vpn-commit.slax

/var/db/scripts/event:

/var/db/scripts/lib:

/var/db/scripts/op:
-rw-r--r--  1 root  wheel      11485 Sep 21  2010 jcs-load-config-op.slax
```

2. Log in to the responding Routing Engine, and verify that the files are synchronized.

```
user@host> request routing-engine login other-routing-engine
user@host1> file list /var/db/scripts/* detail
/var/db/scripts/commit:
-rw-r--r--  1 remote wheel      1014 Jul 17 16:18 vpn-commit.slax

/var/db/scripts/event:

/var/db/scripts/lib:

/var/db/scripts/op:
-rw-r--r--  1 root  wheel      11485 Sep 21  2010 jcs-load-config-op.slax
```

**Meaning** The scripts present on each Routing Engine are identical indicating that the device successfully synchronized the scripts from the requesting Routing Engine to the responding Routing Engine.

## Troubleshooting

### Troubleshooting Script Synchronization Failure

---

**Problem** The device does not synchronize the scripts present on the requesting Routing Engine to the other Routing Engine.

**Solution** Verify the following:

- You configured the **synchronize** statement at the **[edit system scripts]** hierarchy level.
- You are viewing the correct directories on each Routing Engine.

If the **load-scripts-from-flash** statement is configured for the requesting Routing Engine, the device synchronizes scripts from flash memory on the requesting Routing Engine to flash memory on the responding Routing Engine.

- You executed a **commit synchronize** command when committing the configuration.

The device does not synchronize scripts for a **commit** operation, only for a **commit synchronize** operation.

- The commit check and commit operations for the requesting Routing Engine are successful.

If the commit check operation for the requesting Routing Engine fails, the process stops, and the scripts are not copied to the responding Routing Engine.

- Related Documentation**
- [Synchronizing Scripts Between Routing Engines on page 269](#)
  - [synchronize on page 265](#)



## CHAPTER 11

# Configuring the Session Protocol for Scripts

- Specifying the Session Protocol for Connections Using Junos Automation Scripts on page 139

## Specifying the Session Protocol for Connections Using Junos Automation Scripts

- Session Protocol in Junos Automation Scripts Overview on page 139
- Example: Specifying the Session Protocol for a Connection Using an Automation Script on page 141

### Session Protocol in Junos Automation Scripts Overview

The Junos XML management protocol is a Juniper Networks proprietary protocol used to request information from and configure devices running Junos OS. The NETCONF XML management protocol is a standard used to request and change configuration information on a routing, switching, or security device. The NETCONF protocol is defined in [RFC 4741](#), *NETCONF Configuration Protocol*, which is available at <http://www.ietf.org/rfc/rfc4741.txt>.

Starting with Junos OS Release 11.4, the **jcs:open()** function includes the option to create a session either with the Junos XML protocol server on devices running Junos OS or with the NETCONF server on devices where NETCONF service over SSH is enabled. Previously, the function supported only sessions with the Junos XML protocol server on devices running Junos OS. The additional support for NETCONF sessions permits automation scripts to configure and manage devices in a multi-vendor environment.

The **jcs:open()** function supports the following session protocol types:

- **junoscript**—Session with the Junos XML protocol server on a routing, switching, or security device running Junos OS. This session type supports the operations defined in the Junos XML protocol and the Junos XML API, which are used to configure devices running Junos OS or to request information about the device configuration or operation. This is the default session type.
- **netconf**—Session with the NETCONF XML protocol server on a routing, switching, or security device over an SSHv2 connection. The device to which the connection is made must be enabled for NETCONF service over SSH. NETCONF over SSH is described in

[RFC 4742](#), *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, which is available at <http://www.ietf.org/rfc/rfc4742.txt>.

- **junos-netconf**—Proprietary session with the NETCONF XML protocol server over an SSHv2 connection on a routing, switching, or security device running Junos OS.

The NETCONF server on a device running Junos OS has the additional capabilities defined in <http://xml.juniper.net/netconf/junos/1.0>. The NETCONF server on these devices supports NETCONF XML protocol operations, most Junos XML protocol operations, and the tag elements defined in the Junos XML API. For **netconf** and **junos-netconf** sessions with devices running Junos OS, you should use only native NETCONF XML protocol operations and the extensions available in the Junos XML protocol for configuration functions as documented in the *NETCONF XML Management Protocol Developer Guide*.

The syntax for the **jcs:open()** function when specifying a session protocol is:

**SLAX Syntax**      `var $connection = jcs:open(remote-hostname, session-options);`

**XSLT Syntax**      `<xsl:variable name="connection" select="jcs:open(remote-hostname, session-options)"/>`

The *session-options* parameter is an XML node-set that specifies the session type and connection parameters. The session type is one of three values: **junoscript**, **netconf**, or **junos-netconf**. If you do not specify a session type, the default is **junoscript**, which opens a session with the Junos XML protocol server only on devices running Junos OS. The format of the node-set is:

```
var $session-options := {
  <method> ("junoscript" | "netconf" | "junos-netconf");
  <username> "username";
  <passphrase> "passphrase";
  <password> "password";
  <port> "port-number";
  <instance> "routing-instance-name";
  <routing-instance> "routing-instance-name";
}
```

If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script. The **<passphrase>** and **<password>** elements serve the same purpose. If you do not specify a passphrase or password element and it is required for authentication, you should be prompted for one during script execution by the device to which you are connecting.

Optionally, you can specify the server port number for **netconf** and **junos-netconf** sessions. The default NETCONF server port number is 830. If you do not specify a port number for a **netconf** or **junos-netconf** session, **jcs:open()** connects to the NETCONF server using port 830. However, if you specify a port number, **jcs:open()** connects to the given port instead. Specifying a port number has no impact on **junoscript** sessions, which are always established over SSH port 22.

To redirect the SSH connection to originate from within a specific routing instance, include the **instance** or **routing-instance** element and the routing instance name. The routing instance must be configured at the **[edit routing-instances]** hierarchy level. The remote device must be reachable either using the routing table for that routing instance or from

one of the interfaces configured under that routing instance. The **instance** and **routing-instance** elements serve the same purpose.

To verify the protocol for a specific connection, call the **jcs:get-protocol(connection)** extension function and pass the connection handle as the argument. The function returns "junoscript", "netconf", or "junos-netconf", depending on the session type.

During session establishment with a NETCONF server, the client application and NETCONF server each emit a **<hello>** tag element to specify which operations, or *capabilities*, they support from among those defined in the NETCONF specification or published as proprietary extensions. In **netconf** and **junos-netconf** sessions, you can retrieve the session capabilities of the NETCONF server by calling the **jcs:get-hello(connection)** extension function.

For example, the NETCONF server on a typical device running Junos OS might return the following capabilities:

```
<hello>
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
    </capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
    </capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:validate:1.0
    </capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
    </capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>20826</session-id>
</hello>
```

### Example: Specifying the Session Protocol for a Connection Using an Automation Script

The following example demonstrates how to specify the session protocol within a Junos automation script when creating a connection with a remote device. Specifically, the example op script establishes a NETCONF session with a remote device running Junos OS, retrieves and prints the NETCONF server capabilities, and then updates and commits the configuration on that device.

- [Requirements on page 142](#)
- [Overview and Script on page 142](#)
- [Configuration on page 147](#)
- [Verification on page 147](#)
- [Troubleshooting on page 150](#)

## Requirements

---

- Routing, switching, or security device running Junos OS Release 11.4 or later.
- Client application can log in to the device where the NETCONF server resides.
- NETCONF service over SSH is enabled on the device where the NETCONF server resides.

## Overview and Script

---

Starting with Junos OS Release 11.4, the `jcs:open()` function includes the option to create a session either with the Junos XML protocol server on devices running Junos OS or with the NETCONF server on devices where NETCONF service over SSH is enabled. In the following example, the script creates a connection and establishes a NETCONF session with a remote device running Junos OS. If the connection and session are successfully established, the script updates the configuration on the remote device to add the `ftp` statement to the `[edit system services]` hierarchy level. The script also retrieves and prints the session protocol and the capabilities of the NETCONF server.

The script takes one argument, `remote-host`, which is the IP address or hostname of the remote device. The `arguments` variable is declared at the global level of the script so that the argument name and description are visible in the command-line interface (CLI) when a user requires context-sensitive help.

The variable `netconf` is a node-set that specifies the session protocol and the connection parameters for the remote device. The value of the `<method>` element is set to “netconf” to establish a session with the NETCONF server over an SSHv2 connection. The `<username>` element specifies the username for the connection. If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script. In this example, the passphrase and port are not specified. If a passphrase is required for authentication, the remote device should prompt for one during script execution. The script establishes the session using the default NETCONF port 830.

If the connection and establishment of the NETCONF session are successful, the script executes remote procedure calls (RPCs). The RPCs contain the tag elements `<lock>`, `<edit-config>`, `<commit>`, and `<unlock>`, which are NETCONF operations to lock, edit, commit, and unlock the candidate configuration. The script stores the RPC for each task in a separate variable. The results for each RPC are also stored separately and parsed for errors. The script only executes each subsequent step if the previous step is successful. For example, if the script cannot lock the configuration, it does not execute the RPCs to edit, commit, or unlock the configuration.

The variable `rpc-edit-config` contains the tag element `<edit-config>`, which is a NETCONF operation to modify a configuration. The child element, `<config>`, includes the modified portion of the configuration that is merged with the candidate configuration on the device. If errors are encountered, the script calls the `copy-of` statement to copy the result tree fragment variable to the results tree so that the error message prints to the CLI during script execution.

SLAX Syntax      version 1.0;

```

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

var $arguments = {
  <argument> {
    <name> "remote-host";
    <description> "device hostname or IP address to which to connect";
  }
}
param $remote-host;

match / {

  <op-script-results> {

    var $netconf := {
      <method> "netconf";
      <username> "bsmith";
    }

    var $rpc-lock-config = {
      <lock> {
        <target> {
          <candidate>;
        }
      }
    }

    var $rpc-unlock-config = {
      <unlock> {
        <target> {
          <candidate>;
        }
      }
    }

    var $rpc-commit = {
      <commit>;
    }

    var $rpc-edit-config = {
      <edit-config> {
        <target> {
          <candidate>;
        }
        <default-operation> "merge";
        <config> {
          <configuration> {
            <system> {
              <services> {
                <ftp>;
              }
            }
          }
        }
      }
    }
  }
}

```

```

    }
  }
}

if ($remote-host = '') {
  <xnm:error> {
    <message> "missing mandatory argument 'remote-host'";
  }
}
else {

  var $connection = jcs:open($remote-host, $netconf);
  if ($connection) {

    /* request protocol and capabilities */
    var $protocol = jcs:get-protocol($connection);
    var $capabilities = jcs:get-hello($connection);

    <output> "\nSession protocol: " _ $protocol _ "\n";
    copy-of $capabilities;

    /* execute rpcs to lock, edit, commit, and unlock config */
    var $lock-reply = jcs:execute($connection, $rpc-lock-config);
    if ($lock-reply/../../rpc-error) {
      copy-of $lock-reply;
    }
    else {
      var $edit-config-reply = jcs:execute($connection, $rpc-edit-config);
      if ($edit-config-reply/../../rpc-error) {
        <output> "Configuration error: " _ $edit-config-reply/../../error-message/_
          _ "\nConfiguration not committed.\n";
        copy-of $edit-config-reply;
      }
      else {
        var $commit-reply = jcs:execute($connection, $rpc-commit);
        if ($commit-reply/../../rpc-error) {
          <output> "Commit error or warning: " _ $commit-reply/../../error-message/_;
          copy-of $commit-reply;
        }
      }
      var $unlock-reply = jcs:execute($connection, $rpc-unlock-config);
    }

    expr jcs:close($connection);
  }
  else {
    <output> "\nNo connection - exiting script";
  }
}
}
}

```

**XSLT Syntax**

```

<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:junos="http://xml.juniper.net/junos/*/junos"

```

```

xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm"
xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0"
xmlns:ext="http://xmlsoft.org/XSLT/namespace" version="1.0">

<xsl:variable name="arguments">
  <argument>
    <name>remote-host</name>
    <description>device hostname or IP address to which to connect</description>
  </argument>
</xsl:variable>

<xsl:param name="remote-host"/>

<xsl:template match="/">
  <op-script-results>
    <xsl:variable name="netconf-temp-1">
      <method>netconf</method>
      <username>bsmith</username>
    </xsl:variable>
    <xsl:variable xmlns:ext="http://xmlsoft.org/XSLT/namespace"
      name="netconf" select="ext:node-set($netconf-temp-1)"/>

    <xsl:variable name="rpc-lock-config">
      <lock>
        <target>
          <candidate/>
        </target>
      </lock>
    </xsl:variable>

    <xsl:variable name="rpc-unlock-config">
      <unlock>
        <target>
          <candidate/>
        </target>
      </unlock>
    </xsl:variable>

    <xsl:variable name="rpc-commit">
      <commit/>
    </xsl:variable>

    <xsl:variable name="rpc-edit-config">
      <edit-config>
        <target>
          <candidate/>
        </target>
        <default-operation>merge</default-operation>
        <config>
          <configuration>
            <system>
              <services>
                <ftp/>
              </services>
            </system>
          </configuration>

```

```
</config>
</edit-config>
</xsl:variable>

<xsl:choose>
  <xsl:when test="$remote-host = ''">
    <xnm:error>
      <message>missing mandatory argument 'remote-host'</message>
    </xnm:error>
  </xsl:when>
  <xsl:otherwise>
    <xsl:variable name="connection" select="jcs:open($remote-host, $netconf)"/>

    <xsl:choose>
      <xsl:when test="$connection">

        <!-- request protocol and capabilities -->
        <xsl:variable name="protocol" select="jcs:get-protocol($connection)"/>
        <xsl:variable name="capabilities" select="jcs:get-hello($connection)"/>
        <output>
          <xsl:value-of select="concat('&#10;Session protocol: ', $protocol, '&#10;')"/>
        </output>
        <xsl:copy-of select="$capabilities"/>

        <!-- execute rpcs -->
        <xsl:variable name="lock-reply"
          select="jcs:execute($connection, $rpc-lock-config)"/>
        <xsl:choose>
          <xsl:when test="$lock-reply/../../rpc-error">
            <xsl:copy-of select="$lock-reply"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:variable name="edit-config-reply"
              select="jcs:execute($connection, $rpc-edit-config)"/>
            <xsl:choose>
              <xsl:when test="$edit-config-reply/../../rpc-error">
                <output>
                  <xsl:value-of select="concat('Configuration error: ',
                    $edit-config-reply/../../error-message/,
                    '&#10;Configuration not committed.&#10;')"/>
                </output>
                <xsl:copy-of select="$edit-config-reply"/>
              </xsl:when>
              <xsl:otherwise>
                <xsl:variable name="commit-reply"
                  select="jcs:execute($connection, $rpc-commit)"/>
                <xsl:if test="$commit-reply/../../rpc-error">
                  <output>
                    <xsl:value-of select="concat('Commit error or warning: ',
                      $commit-reply/../../error-message/)"/>
                  </output>
                  <xsl:copy-of select="$commit-reply"/>
                </xsl:if>
              </xsl:otherwise>
            </xsl:choose>
            <xsl:variable name="unlock-reply" select="jcs:execute($connection,
```



```

        $rpc-unlock-config)"/>
    </xsl:otherwise>
</xsl:choose>

    <xsl:value-of select="jcs:close($connection)"/>
</xsl:when>
<xsl:otherwise>
    <output>No connection - exiting script</output>
</xsl:otherwise>
</xsl:choose>
</xsl:otherwise>
</xsl:choose>
</op-script-results>
</xsl:template>
</xsl:stylesheet>

```

## Configuration

### Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the XSLT or SLAX script into a text file, name the file **netconf-session.xml** or **netconf-session.slax** as appropriate, and copy it to the **/var/db/scripts/op/** directory on the device.
2. In configuration mode, include the **file** statement at the **[edit system scripts op]** hierarchy level and **netconf-session.xml** or **netconf-session.slax** as appropriate.

```

[edit system scripts op]
bsmith@local-host# set file netconf-session.(slax | xml)

```

3. Issue the **commit and-quit** command.

```

[edit]
bsmith@local-host# commit and-quit

```

4. Execute the op script on the local device by issuing the **op netconf-session** operational mode command and include any necessary arguments.

In this example, the user, bsmith, is connecting to the remote device, fivestar. The remote device has dual routing engines, so the **commit** operation returns a warning that the **commit synchronize** command should be used to commit the new candidate configuration to both routing engines.

```

bsmith@local-host> op netconf-session remote-host fivestar
bsmith@fivestar's password:
Session protocol: netconf
Commit error or warning:
graceful-switchover is enabled, commit synchronize should be used

```

## Verification

Confirm that the device is working properly.

- [Verifying Op Script Execution on page 148](#)
- [Verifying the Configuration Changes on page 149](#)

*Verifying Op Script Execution*

**Purpose** Verify that the script behaves as expected.

**Action** Review the script output in the CLI and in the op script log file. Take particular note of any errors that occurred during execution. The default op script log file is `/var/log/op-script.log`. If the log file is significantly lengthy, limit the display by appending the `| last number-of-lines` option to the `show log` command and specify the number of lines to print to the CLI. The output within the `<op-script-results>` element is relevant to the script execution.

```
bsmith@local-host> show log op-script.log | last 100
...output omitted for brevity...
<op-script-results xmlns:junos="http://xml.juniper.net/junos/*/junos"
xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm" xml
ns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0"
xmlns:ext="http://xmlsoft.org/XSLT/namespace">
<output>
Session protocol: netconf
</output>
<hello>
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>

    <capability>
      urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
    </capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>29087</session-id>
</hello>
  <output>Commit error or warning:
graceful-switchover is enabled, commit synchronize should be used
</output>
  <rpc-error>
<error-severity>warning</error-severity>
<error-message>
graceful-switchover is enabled, commit synchronize should be used
</error-message>
</rpc-error>
  <ok/>
</op-script-results>
```

You can also obtain more descriptive script output on a device running Junos OS by including the `| display xml` option when you execute an op script.

```
bsmith@local-host> op netconf-session remote-host fivestar | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/11.4D0/junos">
  <output>
    Session protocol: netconf
  </output>
  <hello>
```

```

<capabilities>
  <capability>
    urn:ietf:params:xml:ns:netconf:base:1.0
  </capability>
  <capability>
    urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
  </capability>
  <capability>
    urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
  </capability>
  <capability>
    urn:ietf:params:xml:ns:netconf:capability:validate:1.0
  </capability>
  <capability>
    urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
  </capability>
  <capability>
    http://xml.juniper.net/netconf/junos/1.0
  </capability>
  <capability>
    http://xml.juniper.net/dmi/system/1.0
  </capability>
</capabilities>
<session-id>
  29087
</session-id>
</hello>
<output>
  Commit error or warning:
  graceful-switchover is enabled, commit synchronize should be used
</output>
<rpc-error>
  <error-severity>
    warning
  </error-severity>
  <error-message>
    graceful-switchover is enabled, commit synchronize should be used
  </error-message>
</rpc-error>
<ok/>
</op-script-results>
<cli>
  <banner></banner>
</cli>
</rpc-reply>

```

**Meaning** This example creates a NETCONF session on a remote device running Junos OS. The capabilities of the NETCONF server include both standard NETCONF operations and Juniper Networks proprietary extensions, which are defined in <http://xml.juniper.net/netconf/junos/1.0> and <http://xml.juniper.net/dmi/system/1.0>. The RPC results for the **commit** operation include one warning, but the commit operation is still successful.

### *Verifying the Configuration Changes*

**Purpose** Verify that the commit was successful by viewing the configuration change and the commit log on the remote device.

**Action** On the remote device, execute the **show configuration system services** operational mode command to view the **[edit system services]** hierarchy level of the configuration. If the script is successful, the configuration includes the **ftp** statement.

```
bsmith@fivestar> show configuration system services
ftp;
netconf {
    ssh;
}
```

Additionally, you can review the commit log. On the remote device, execute the **show system commit** operational mode command to view the commit log. In this example, the log confirms that bsmith committed the candidate configuration in a NETCONF session at the given date and time.

```
bsmith@fivestar> show system commit
0   2011-07-11 12:04:01 PDT by bsmith via netconf
1   2011-07-08 15:16:33 PDT by root via cli
```

---

### Troubleshooting

- [Troubleshooting Connection Errors on page 150](#)
- [Troubleshooting Configuration Lock Errors on page 151](#)
- [Troubleshooting Configuration Syntax Errors on page 152](#)

#### ***Troubleshooting Connection Errors***

**Problem** The script generates the following error message:

```
hello packet:1:(0) Document is empty
hello packet:1:(0) Start tag expected, '<' not found
error: netconf: could not read hello
error: did not receive hello packet from server
error: Error in creating the session with "fivestar" server
No connection - exiting script
```

Potential causes for the connection error include:

- The device or interface to which you are connecting is down or unavailable.
- The script argument for the IP address or DNS name of the remote device is incorrect.
- The connection timeout value was exceeded before establishing the connection.
- The user authentication for the remote device is not valid or is entered incorrectly.
- You are trying to establish a NETCONF session, and NETCONF over SSH is not enabled on the device where the NETCONF server resides, or it is enabled on a different port.

**Solution** Ensure that the remote device is up and running and that the user has access to the device. Also verify that you supplied the correct argument for the IP address or DNS name of the remote device when executing the script.

For NETCONF sessions, ensure that you have enabled NETCONF over SSH on the device where the NETCONF server resides. Since the example program does not specify a specific

port number for the NETCONF session, the session is established on the default NETCONF-over-SSH port, 830. To verify whether NETCONF over SSH is enabled on the default port for a device running Junos OS, enter the following operational mode command on the remote device:

```
bsmith@fivestar> show configuration system services
```

```
netconf {
  ssh;
}
```

If the **netconf** configuration hierarchy is absent on the remote device, issue the following statements in configuration mode to enable NETCONF over SSH on the default port:

```
[edit]
bsmith@fivestar# set system services netconf ssh
bsmith@fivestar# commit
```

If the **netconf** configuration hierarchy specifies a port other than the default port, include the port number in the XML node-set that you pass to the **jcs:open()** function. For example, the following device is configured for NETCONF over SSH on port 12345:

```
bsmith@fivestar> show configuration system services
netconf {
  ssh {
    port 12345;
  }
}
```

To create a NETCONF session on the alternate port, include the new port number in the XML node-set.

```
var $netconf := {
  <method> "netconf";
  <username> "bsmith";
  <port> "12345";
}
var $connection = jcs:open($remote-host, $netconf);
...
```

### ***Troubleshooting Configuration Lock Errors***

**Problem** The script generates one of the following error messages:

```
configuration database locked by:
  root terminal p0 (pid 24113) on since 2011-07-11 11:48:06 PDT, idle 00:07:59

Users currently editing the configuration:
  root terminal p1 (pid 24279) on since 2011-07-11 12:28:30 PDT
  {master}[edit]

configuration database modified
```

**Solution** Another user currently has a lock on the candidate configuration or has modified the candidate configuration but has not yet committed the configuration. Wait until the lock is released, and then execute the program.

### *Troubleshooting Configuration Syntax Errors*

**Problem** The following error message prints to the CLI:

```
Configuration error: syntax error
Configuration not committed.
```

Examine the result tree for additional information. In this case, the result tree shows the following error message:

```
<rpc-error>
  <error-severity>
    error
  </error-severity>
  <error-info>
    <bad-element>
      ftp2
    </bad-element>
  </error-info>
  <error-message>
    syntax error
  </error-message>
</rpc-error>
```

**Solution** The **<bad-element>** tag element indicates that the configuration statement is not valid. Correct the configuration hierarchy and run the script. In this example error, the user entered the tag **<ftp2>** instead of **<ftp>**. Since that is not an acceptable element in the configuration, the NETCONF server returns an error.

**Related  
Documentation**

- [Session Protocol in Junos Automation Scripts Overview on page 139](#)
- *Junos XML Management Protocol Developer Guide*
- *NETCONF XML Management Protocol Developer Guide*
- [get-hello\(\) Function \(jcs Namespace\) on page 24](#)
- [get-protocol\(\) Function \(jcs Namespace\) on page 25](#)
- [open\(\) Function \(jcs Namespace\) on page 28](#)

## CHAPTER 12

# Configuring Limits on Event Policies and Memory Allocation for Scripts

- [Example: Configuring Limits on Executed Event Policies and Memory Allocation for Scripts on page 153](#)

## Example: Configuring Limits on Executed Event Policies and Memory Allocation for Scripts

---

- [Overview of Limits on Executed Event Policies and Memory Allocation for Scripts on page 153](#)
- [Example: Configuring Limits on Executed Event Policies and Memory Allocation for Scripts on page 154](#)

## Overview of Limits on Executed Event Policies and Memory Allocation for Scripts

By default, the maximum number of event policies that can run concurrently in the system is 15, and the maximum amount of memory allocated for the data segment portion of an executed script is half of the total available memory of the system, up to a maximum value of 128 MB. If a script requires more memory during execution than the set maximum limit, the script exits. If the system is running the maximum number of event policies, the system ignores any triggered event policy until such time that another policy finishes. The system logs the `EVENTD_POLICY_LIMIT_EXCEEDED` message for any triggered event policies that were not executed.

Starting with Junos OS Release 12.3, you can configure limits on the maximum number of concurrently running event policies and the maximum amount of memory allocated for the data segment for scripts of a given type.

Depending on the device and its function in the network, it might be necessary to configure larger or smaller limits on the number of event policies that can execute concurrently and the maximum amount of memory allocated to scripts. You might configure smaller limits on critical devices to ensure that priority processes are not adversely impacted, and that the device can perform all necessary functions in the network. Additionally, during normal device operation, you might want to allocate disproportionate amounts of memory to different script types. For example, a device might have a particular type of script that plays a vital role in its operation and requires a specific amount of memory to ensure proper execution.

To set the maximum number of event policies that can run concurrently on a device, configure the **max-policies** *policies* statement at the **[edit event-options]** hierarchy level. You can configure a maximum of 0 through 20 policies.

```
[edit]
event-options {
  max-policies policies;
}
```

To set the maximum memory allocated to the data segment for scripts of a given type, configure the **max-datasize** *size* statement under the hierarchy appropriate for that script type, where *size* is the memory in bytes. To specify the memory in kilobytes, megabytes, or gigabytes, append **k**, **m**, or **g**, respectively, to the size. You can configure the memory in the range from 2,3068,672 bytes (22 MB) through 1,073,741,824 bytes (1 GB).

```
[edit]
event-options {
  event-script {
    max-datasize size;
  }
}
system {
  scripts {
    commit {
      max-datasize size;
    }
    op {
      max-datasize size;
    }
  }
}
```

When the **max-datasize** statement is configured and a script executes, Junos OS sets the maximum memory limit for that script to the configured value irrespective of the total memory available on the system at the time of execution. If the script exceeds the maximum memory limit during execution, it exits gracefully.

### Example: Configuring Limits on Executed Event Policies and Memory Allocation for Scripts

This example configures a maximum value for the number of event policies that the device can execute concurrently and a maximum memory limit for executed commit, event, and op scripts.

- [Requirements on page 154](#)
- [Overview on page 155](#)
- [Configuration on page 155](#)
- [Verification on page 156](#)

---

#### Requirements

A device running Junos OS Release 12.3 or later.



## Overview

This example configures a device running Junos OS to limit the number of event policies that can run simultaneously on that device to a maximum of 12 policies. Additionally, the example configures each script type with a maximum amount of memory that the system can allocate to the data segment portion of a script of that type. The device is configured to allocate 192 MB for each executed commit script and event script and 100 MB for each executed op script.

## Configuration

### CLI Quick Configuration

To quickly configure this example, copy the following commands, paste them in a text file, remove any line breaks, change any details necessary to match your network configuration, and then copy and paste the commands into the CLI at the **[edit]** hierarchy level:

```
set system scripts commit max-datasize 192m
set system scripts op max-datasize 100m
set event-options max-policies 12
set event-options event-script max-datasize 192m
```

### Step-by-Step Procedure

1. Configure the maximum number of event policies that can execute concurrently.  

```
[edit]
user@host# set event-options max-policies 12
```
2. Configure the maximum memory allocated for the data segment for each executed commit script.  

```
[edit]
user@host# set system scripts commit max-datasize 192m
```
3. Configure the maximum memory allocated for the data segment for each executed op script.  

```
[edit]
user@host# set system scripts op max-datasize 100m
```
4. Configure the maximum memory allocated for the data segment for each executed event script.  

```
[edit]
user@host# set event-options event-script max-datasize 192m
```
5. Commit the configuration.  

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

## Results

```
[edit]
event-options {
  event-script {
    max-datasize 192m;
  }
}
```

```
    max-policies 12;
  }
  system {
    scripts {
      commit {
        max-datasize 192m;
      }
      op {
        max-datasize 100m;
      }
    }
  }
}
```

---

### Verification

Confirm that the configuration is working properly.

#### *Verifying the Limit on Concurrently Executing Event Policies*

**Purpose** If the system is running the maximum number of event policies, the system ignores any triggered event policy until such time that another policy finishes. The system logs the EVENTD\_POLICY\_LIMIT\_EXCEEDED message for any triggered event policies that were not executed. By default, system log messages are recorded in the **messages** log file.

**Action** Review the configured log file to verify whether any policies were barred from execution, because the maximum limit was reached. You can narrow the output to include only the relevant error messages by appending **| match EVENTD\_POLICY\_LIMIT\_EXCEEDED**.

```
user@R1> show log messages | match EVENTD_POLICY_LIMIT_EXCEEDED
Jun 11 17:02:42 R1 eventd[1177]: EVENTD_POLICY_LIMIT_EXCEEDED: Unable to execute
policy 'raise-trap' because current number of policies (12) exceeds system limit
(12)
[output omitted]
```

**Related Documentation**

- [max-datasize on page 264](#)
- [max-policies](#)

## CHAPTER 13

# Provisioning Services Using Service Template Automation

- [Example: Service Template Automation on page 157](#)

### Example: Service Template Automation

---

- [Service Template Automation Overview on page 157](#)
- [Example: Configuring Service Template Automation on page 158](#)

### Service Template Automation Overview

Starting with Junos OS Release 12.3, you can use service template automation (STA) to provision services such as VPLS VLAN, Layer 2 and Layer 3 VPNs, and IPsec across similar platforms running Junos OS. Service template automation uses the **service-builder.slax** op script to transform a user-defined service template definition into a uniform API, which you can then use to configure and provision services on similar platforms running Junos OS. This permits you to create a service template on one device, generalize the parameters, and then quickly and uniformly provision that service on other devices. This decreases the time required to configure the same service on multiple devices, and reduces configuration errors associated with manually configuring each device.

The following process outlines how to use service template automation to provision services:

1. Create a service template definition.
2. Execute the **service-builder.slax** script and define service-specific instance parameters.
3. Generate the service interface, which automatically builds the required interface (API) from the template.
4. Enable the service interface on each device where the service is required.
5. Provision systems by invoking the service interface using NETCONF and supplying the service parameter values.

You create a new service template by configuring the hierarchies for the actual service to be provisioned on a device running Junos OS. Service template hierarchies are configured at the **[edit groups]** hierarchy level. When creating the service template:

- Do not include **apply-groups** or **apply-macro** statements.
- Do not include any statements that are supported on the current device that are not also supported on the devices where the service will be provisioned (for example dual Routing Engine versus single Routing Engine).
- Commit the configuration. The service template group configuration is read from the committed configuration.

Once you create the basic service template definition, you invoke the **service-builder.slax** op script. The script reads the service template information from the committed configuration and uses an interactive interface to help you build and generate the service API. You have the option to parameterize every variable in the service template or only selected variables. For each selected variable, you create a generic service template parameter. The **service-builder.slax** script guides you through the creation and configuration of each parameter.

After you define the service template parameters, you generate the service interface. This creates a platform-specific service op script. If the **load-scripts-from-flash** statement is configured, the generated service script is stored in the **/config/scripts/op** directory in flash memory. Otherwise, the generated script is stored in the **/var/db/scripts/op** directory on the hard disk.

To enable the service interface on a device, you enable the generated service script in the configuration as you would any op script. You can enable the service interface on the local device using the **service-builder.slax** script or by manually updating the configuration. To enable the service interface on a similar platform, you must copy the generated service script to the corresponding directory on the new device and enable the service script in the configuration.

To provision the service on a device, invoke the service interface using NETCONF, and supply the necessary values for each parameter. Alternatively, you can invoke the service interface in the CLI by executing the service script and supplying the necessary values for each parameter as command-line arguments to the script. You can direct the service script to create a new service configuration, or update or delete an existing service configuration. The service script makes the changes to the candidate configuration and then commits the configuration. The service script does not support the context-sensitive help and auto-completion features available in the Junos OS CLI.

## Example: Configuring Service Template Automation

This example shows how to use service template automation to provision services across similar platforms running Junos OS.

- [Requirements on page 159](#)
- [Overview on page 159](#)
- [Configuration on page 160](#)

- [Verification on page 168](#)
- [Troubleshooting on page 169](#)

## Requirements

---

- Two MX Series devices running Junos OS Release 12.3 or later.

## Overview

---

This example uses service template automation to provision services on an MX Series router. To use the service template automation **service-builder.slax** script, you must first copy the script to the `/var/db/scripts/op` or `/config/scripts/op` directory and enable the script on the device.

The following process outlines how to use service template automation to provision services:

1. Create a service template definition.
2. Execute the **service-builder.slax** script, and define service-specific instance parameters.
3. Generate the service interface.
4. Enable the service interface on each device where the service is required.
5. Provision systems by invoking the service interface using NETCONF and supplying the service parameter values.

This example creates a new VPN service interface on an MX Series device running Junos OS Release 12.3 and provisions the service on a second MX Series device running Junos OS Release 12.3. You configure service template definitions under the **[edit groups]** hierarchy level. For this example, the service name is **vpn-service**, and the template group name is **vpn-service-template-group**. The **load merge terminal** configuration mode command loads the service template configuration hierarchies into the candidate configuration, which is then committed.

Once you create the initial service template, you execute the **service-builder.slax** script. The script prompts for the service name and the template group name, and then reads the service template configuration from the committed configuration.

The **service-builder.slax** script interface consists of two menus: **Main Menu** and **Hierarchies Menu**. Within the **Main Menu**, you can review the variables defined in the service template configuration, or you can build or enable the service API. The **Build Service API** menu option displays the **Hierarchies Menu**, which steps you through the parameterization of the variables. The default is to parameterize every variable, or you can choose to parameterize selected variables. If you must exit the **service-builder.slax** script while building the service API, you must finish configuring all the parameters for the current hierarchy in order to save that hierarchy configuration when you exit using the **Quit** option. Then you can finish configuring any incomplete hierarchies at a later time. This example parameterizes two variables: the interface name and the interface description. After the parameters are specified, the service builder script generates the service script.

The **Enable Service API** menu option enables the service script on the local device. To enable the service script on the second MX Series device, the generated service script is copied to the `/var/db/scripts/op` directory on the second device, and the script is enabled in the configuration. If the **load-scripts-from-flash** statement is configured, the script must be copied to the corresponding directory on the flash drive instead.

NETCONF is used to provision the service on the remote MX Series device. The NETCONF remote procedure call (RPC) action depends on whether the service is a new service or an existing service. Supported actions include **create**, **update**, and **delete**. This example creates a new service. If the given service is already provisioned on the device and you are updating or deleting the service parameters, you can alter the RPC to perform these actions.

---

### Configuration

- [Storing and Enabling the Service Builder Script on page 160](#)
- [Configuring the Service Template Definition on page 161](#)
- [Configuring and Generating the Service Interface on page 162](#)
- [Verifying the Service Interface on page 164](#)
- [Enabling the Service Interface on page 165](#)
- [Provisioning the Service Using NETCONF on page 166](#)
- [Updating or Deleting Services Using NETCONF on page 167](#)

#### *Storing and Enabling the Service Builder Script*

#### **Step-by-Step Procedure**

The Junos OS installation includes the **service-builder.slax** script, which is stored in the `/usr/libexec/scripts/op/` directory on the device. To use the **service-builder.slax** script, you must first copy it to the `op` scripts directory and enable it in the configuration. Only users in the Junos OS superuser login class can access and edit files in these directories.

1. Copy the **service-builder.slax** script to the `/var/db/scripts/op` directory on the hard disk or the `/config/scripts/op` directory on the flash drive.

```
user@host> file copy /usr/libexec/scripts/op/service-builder.slax /var/db/scripts/op
```

2. Verify that the script is in the correct directory by using the **file list** operational mode command.

```
user@host> file list /var/db/scripts/op
/var/db/scripts/op:
service-builder.slax*
```

3. Enable the script in the configuration.

```
[edit]
user@host# set system scripts op file service-builder.slax
```

4. If you store scripts in and load them from flash, configure the **load-scripts-from-flash** statement, if it is not already configured.

```
[edit]
user@host# set system scripts load-scripts-from-flash
```

5. Commit the configuration.

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

### *Configuring the Service Template Definition*

**Step-by-Step Procedure** To create a new service template on a device running Junos OS:

1. Select a service name.

This example uses **vpn-service**.

2. In configuration mode, create a new group, which will contain the hierarchies for the actual service to be provisioned.

```
[edit]
user@host# set groups vpn-service-template-group
```

3. Configure the hierarchies for the service.

For this example, the pre-constructed configuration hierarchies are loaded into the candidate configuration using the **load merge terminal** command.

```
[edit]
user@host# load merge terminal
groups {
  vpn-service-template-group {
    interfaces {
      ge-2/2/6 {
        description "connected to customer3-site-1";
        unit 0 {
          family bridge {
            interface-mode access;
            vlan-id 300;
          }
        }
      }
    }
    protocols {
      rstp {
        interface ge-2/3/0;
      }
      mvrp {
        interface ge-2/3/0;
      }
    }
    bridge-domains {
      bd {
        vlan-id-list 100;
      }
    }
  }
}
[Ctrl+D]
```

4. Verify that the configuration syntax is correct.

```
[edit]
user@host# commit check
configuration check succeeds
```

5. Commit the configuration.

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

### *Configuring and Generating the Service Interface*

#### **Step-by-Step Procedure**

To configure and generate the service interface:

1. In operational mode, execute the **service-builder.slax** script, which starts an interactive Service Builder session.

```
user@host> op service-builder
Welcome to Service Builder Script: (v1.0)
-
Enter the service name :
```

2. Enter the service name that was defined in [“Configuring the Service Template Definition” on page 161](#).

```
Enter the service name : vpn-service
```

3. Enter the group name under which the service hierarchies are configured.  
  
This example uses the group name **vpn-service-template-group**. The script reads the configuration specified in the **vpn-service-template-group** hierarchy and then displays the main menu.

```
Enter the group name : vpn-service-template-group
.. reading [edit group vpn-service-template-group] ..
```

```
[Op Script Builder - Main Menu]
-----
1. Show Variables
2. Build Service API
3. Enable Service API
Q. Quit
-----
Enter Selection:>
```

4. (Optional) To review the service template variables that you can parameterize, select the **Show Variables** option.

The script translates the template definition in the candidate configuration into a general parameter list grouped by hierarchy level.

```
[Op Script Builder - Main Menu]
-----
1. Show Variables
2. Build Service API
3. Enable Service API
```



Q. Quit

Enter Selection:> 1

List of variables under each hierarchy to parameterize:

- 
- 1. [ edit groups vpn-service-template-group interfaces ]
  - 1.1. interface/name
  - 1.2. interface/description
  - 1.3. interface/unit/name
  - 1.4. interface/unit/family/bridge/interface-mode
  - 1.5. interface/unit/family/bridge/vlan-id
- 
- 2. [ edit groups vpn-service-template-group protocols ]
  - 2.1. rstp/interface/name
  - 2.2. mvrp/interface/name
- 
- 3. [ edit groups vpn-service-template-group bridge-domains ]
  - 3.1. domain/name
  - 3.2. domain/vlan-id-list
- 

5. To build the Service API, select the **Build Service API** option.

[Op Script Builder - Main Menu]

- 1. Show Variables
- 2. Build Service API
- 3. Enable Service API
- Q. Quit

Enter Selection:> 2

6. From the **Hierarchies Menu**, enter the menu selections for the hierarchies that have variables you want to parameterize, or press Enter to select all hierarchies.

[Op Script Builder - Hierarchies Menu]

- 1. interfaces
- 2. protocols
- 3. bridge-domains
- Q. Quit

Please enter multiple selections separated by a comma (,) only.

Enter Selection:> [default:all] 1

7. From the variables list, enter the menu selections for the variables you want to parameterize for the service interface, or press Enter to parameterize all variables within that hierarchy.

List of variables to parameterize: ...

[ edit groups vpn-service-template-group interfaces ]

- 1. interfaces/interface/name
- 2. interfaces/interface/description
- 3. interfaces/interface/unit/name
- 4. interfaces/interface/unit/family/bridge/interface-mode
- 5. interfaces/interface/unit/family/bridge/vlan-id

```
Q. Quit
Please enter multiple selections separated by a comma (,) only.
-----
Enter Selection:> [default:all] 1,2
```

8. Configure the selected parameters.

The system prompts for the required information. This example configures the interface name parameter as **ifname** and the interface description parameter as **ifdesc**.

```
Enter parameter name for: 1.interfaces/interface/name
*****
[ edit groups vpn-service-template-group interfaces ]
Name for this parameter? ifname
Do you want to revise 'ifname'? (yes/no)[no]: no
Enter parameter name for: 2.interfaces/interface/description
*****
[ edit groups vpn-service-template-group interfaces ]
Name for this parameter? ifdesc
Do you want to revise 'ifdesc'? (yes/no)[no]: no
```

9. Configure the selected parameters at each hierarchy level.

The script iterates over each selected hierarchy and the specified parameters. If you must exit the **service-builder.slax** script while building the service API, you must finish configuring all the parameters for the current hierarchy in order to save that hierarchy configuration when you exit using the **Quit** option.

10. Generate the service interface, which creates the service script.

Once all parameters are configured, the script automatically prompts you to generate the service interface. Press Enter or type yes to generate the service interface.

```
Do you want to commit the previously selected options to create vpn-service
script? (yes/no)[yes]: yes
Created service script: /var/db/scripts/op/vpn-service.slax
```

### ***Verifying the Service Interface***

**Purpose** Verify the creation of the service script. If the **load-scripts-from-flash** statement is configured, the generated file is stored in flash memory. Otherwise, the generated file is stored on the hard disk.

**Action** Issue the **file list** operational mode command. For this example, the **vpn-service.slax** script should be present in the **/var/db/scripts/op** directory. The **service-builder.slax** script also generates the **utility.slax** script in the **/var/db/scripts/op** directory and the **vpn-service-builder-info.xml** file in the **/var/db/scripts/lib** directory. These files are used by the **service-builder.slax** script and should not be deleted.

```
user@host> file list /var/db/scripts/op
/var/db/scripts/op:
service-builder.slax
utility.slax
vpn-service.slax

user@host> file list /var/db/scripts/lib
/var/db/scripts/lib:
vpn-service-builder-info.xml
```

### *Enabling the Service Interface*

**Step-by-Step Procedure** To enable the service interface on a remote device:

1. Copy the generated service script to the device where you are provisioning the new service.

If the **load-scripts-from-flash** statement is not configured, copy the service script to the **/var/db/scripts/op** directory on the second device. Otherwise, the script must be copied to the corresponding directory on the flash drive instead.

Copy the service script to the **/var/db/scripts/op** directory on the second device.

2. Enable the op script in the configuration.

```
[edit]
user@host2# set system scripts op file vpn-service.slax
```

3. Commit the configuration.

```
[edit]
user@host2# commit
commit complete
```

4. In operational mode, verify that the script is enabled and that the service parameters display as arguments for the script.

```
user@host2> op vpn-service ?
Possible completions:
<[Enter]>   Execute this command
<name>      Argument name
action      Please enter either create/delete/update
detail      Display detailed output
ifdesc      Text description of interface
ifname      Name of interface
service-id  Service Name
|           Pipe through a command
```

### *Provisioning the Service Using NETCONF*

#### **Step-by-Step Procedure**

To provision the service:

1. If it is not already configured, configure NETCONF service over SSH on any devices where you are provisioning the new service.

```
[edit]
user@host2# set system services netconf ssh
user@host2# commit
```

2. From a configuration management server, establish a NETCONF session with the device where you are provisioning the service.

```
%ssh -p 830 -s user@host2 netconf
user@host2's password:
```

```
<!-- user user, class super-user -->
<hello>
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
    </capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
    </capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:validate:1.0
    </capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
    </capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>28898</session-id>
</hello>
]]>]]>
```

3. If you are provisioning a new service on the device, enter a remote procedure call (RPC) that calls the service op script using the **create** action, and include values for all parameters that require configuring.

The value for the **service-id** parameter should be identical to the service name.

```
<rpc>
  <op-script>
    <script>vpn-service</script>
    <action>create</action>
    <service-id>vpn-service</service-id>
    <ifname>ge-2/0/5,ge-2/0/6</ifname>
    <ifdesc>connected to customer1-site-1,connected to customer3-site-2</ifdesc>

  </op-script>
</rpc>
```

*Updating or Deleting Services Using NETCONF***Step-by-Step  
Procedure**

To update or delete an existing service:

1. If it is not already configured, configure NETCONF service over SSH on any devices where you are updating or deleting the service.

```
[edit]
user@host2# set system services netconf ssh
user@host2# commit
```

2. From a configuration management server, establish a NETCONF session with the device where you are provisioning the service.

```
%ssh -p 830 -s user@host2 netconf
user@host2's password:
```

```
<!-- user user, class super-user -->
<hello>
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
    </capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
    </capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:validate:1.0
    </capability>
    <capability>
      urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
    </capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>28898</session-id>
</hello>
]]>]]>
```

3. If the given service is already provisioned on the device and you are updating the service, enter an RPC that calls the service op script using the **update** action, and include values for all parameters that require updating.

```
<rpc>
  <op-script>
    <script>vpn-service</script>
    <action>update</action>
    <service-id>vpn-service</service-id>
    <ifname>ge-2/0/5</ifname>
    <ifdesc>connected to customer1-site-2</ifdesc>
  </op-script>
</rpc>
```

4. If the given service is already provisioned on the device and you are deleting some or all of the service parameters, enter an RPC that calls the service op script using the **delete** action, and include any parameters that need to be deleted.

```
<rpc>
  <op-script>
    <script>vpn-service</script>
    <action>update</action>
    <service-id>vpn-service</service-id>
    <ifname>ge-2/0/6</ifname>
  </op-script>
</rpc>
```

---

### Verification

Confirm that the configuration is updated.

- [Verifying the Service Configuration on page 168](#)
- [Verifying the Service Configuration on page 168](#)

#### *Verifying the Service Configuration*

**Purpose** Verify that the commit is successful.

**Action** Issue the **show system commit** operational mode command to view the recent commits. The most recent commit entry shows that a commit was made through the NETCONF server by **user**.

```
user@host2> show system commit

0   2012-05-21 12:15:08 PDT by user via junoscript
1   2012-05-18 09:47:40 PDT by user via other
...
```

#### *Verifying the Service Configuration*

**Purpose** Verify that the service configuration is present in the active configuration.

**Action** Issue the **show configuration | compare rollback *num*** operational mode command to view configuration changes.

```
user@host2> show configuration | compare rollback 1

[edit interfaces]
+   ge-2/0/5 {
+       description "connected to customer1-site-1";
+   }
+   ge-2/0/6 {
+       description "connected to customer3-site-2";
+   }
```

**Meaning** A comparison of the current configuration with the previous configuration shows that the interfaces and interface descriptions were added to the configuration.

## Troubleshooting

---

- [Troubleshooting a Failed Commit on page 169](#)
- [Troubleshooting a Failed Attempt to Delete Service Parameters on page 169](#)

### *Troubleshooting a Failed Commit*

**Problem** You see the following message when creating, updating, or deleting a service on a device through a NETCONF session:

```
<output>
configuration database modified
</output>
```

The configuration has previously uncommitted changes, and the service script cannot commit the service configuration changes.

**Solution** Commit the previous changes or roll back the configuration as appropriate, and then resubmit the service configuration changes.

### *Troubleshooting a Failed Attempt to Delete Service Parameters*

**Problem** You see the following message when deleting a service parameter on a device through NETCONF:

```
<xnm:error xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
  <source-daemon>
    op-script
  </source-daemon>
  <message>
    xsi:attribute: Cannot add attributes to an element if children have been
    already added to the element.
  </message>
</xnm:error>
```

**Solution** The RPC might include both the parameter and a child element. Remove the child element from the RPC.





## CHAPTER 14

# Using libslax

- Downloading and Installing the libslax Distribution on page 171
- libslax Default Extension Libraries: bit, cURL, and xutil on page 171
- SLAX Debugger, Profiler, and callflow on page 181
- Using the SLAX Processor (slaxproc) on page 187

### Downloading and Installing the libslax Distribution

---

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

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

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

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

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

#### Related Documentation

- libslax Distribution Overview on page 109
- libslax Library and Extension Libraries Overview on page 110
- Understanding the SLAX Processor (slaxproc) on page 111

### libslax Default Extension Libraries: bit, cURL, and xutil

---

- libslax bit Extension Library on page 172
- libslax cURL Extension Library on page 174
- libslax xutil Extension Library on page 181

## libslax bit Extension Library

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

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

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

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

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

match / {
  <out> {
    <bit-and> {
      <a1> bit:and("101100", "100101");
      <a2> bit:and($a, $b);
      <a3> bit:and($a, number($b));
    }
    <bit-or> {
      <a1> bit:or("101100", "100101");
      <a2> bit:or($a, $b);
      <a3> bit:or($a, number($b));
    }
    <bit-mask> {
      <a1> bit:mask(0);
      <a2> bit:mask(8, 32);
    }
    <ops> {
      <a1> bit:to-int("10101");
    }
  }
}
```

Table 16 on page 172 lists the functions available in the bit extension library and which are supported in SLAX 1.1 scripts.

**Table 16: Functions in the bit Extension Library**

Function and Arguments	Description	Example
bit:and(b1, b2)	Return the logical AND of two bit strings.	bit:and("101100", "100101") return value: "100100"
bit:clear(b1, bitnum)	Set the specified bit in the bit string to zero and return the new bit string. Bits are numbered starting from zero. If the integer argument is greater than the bit string length, the bit string is extended.	bit:clear("11111", 0) return value: "11110"  bit:clear("11111", 6) return value: "0011111"

Table 16: Functions in the bit Extension Library (*continued*)

Function and Arguments	Description	Example
<code>bit:compare(value1, value2)</code>	Compare two values and return an integer less than, equal to, or greater than zero, if the first argument is found to be less than, equal to, or greater than the second argument, respectively.	<code>bit:compare("10000", 16)</code> return value: 0  <code>bit:compare("11111", "10000")</code> return value: 1
<code>bit:from-hex(string, len?)</code>	Return the value of the hex argument as a bit string. The optional second argument pads the bit string with leading 0's until it is the specified length.	<code>bit:from-hex("0x45", 8)</code> return value: "01000101"
<code>bit:from-int(integer, len?)</code>	Return the value of the integer argument as a bit string. The optional second argument pads the bit string with leading 0's until it is the specified length.	<code>bit:from-int(65, 8)</code> return value: "01000001"
<code>bit:mask(count, len?)</code>	Return a bit string with count low-order bits set to one. The optional second argument pads the bit string with leading 0's until it is the specified length.	<code>bit:mask(4, 8)</code> return value: "00001111"
<code>bit:nand(b1, b2)</code>	Return the logical NAND of two bit strings.	<code>bit:nand("101100", "100101")</code> return value: "010010"
<code>bit:nor(b1, b2)</code>	Return the logical NOR of two bit strings.	<code>bit:nor("101100", "100101")</code> return value: "011011"
<code>bit:not(b1)</code>	Return the inversion (NOT) of a bit string.	<code>bit:not("101100")</code> return value: "010011"
<code>bit:or(b1, b2)</code>	Return the logical OR of two bit strings.	<code>bit:or("101100", "100101")</code> return value: "101101"
<code>bit:set(b1, bitnum)</code>	Set the specified bit in the bit string and return the new bit string. Bits are numbered starting from zero. If the integer argument is greater than the bit string length, the bit string is extended.	<code>bit:set("1001", 2)</code> return value: "1101"  <code>bit:set("1001", 6)</code> return value: "1001001"
<code>bit:to-int(b1)</code>	Return the value of the bit string argument as an integer.	<code>bit:to-int("101100")</code> return value: 44
<code>bit:to-hex(b1)</code>	Return the value of the bit string argument as a string representation of the hex value.	<code>bit:to-hex("101100")</code> return value: "0x2c"

Table 16: Functions in the bit Extension Library (*continued*)

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

## libslax cURL Extension Library

- [Understanding the cURL Extension Library on page 174](#)
- [curl:close on page 177](#)
- [curl:open on page 177](#)
- [curl:perform on page 177](#)
- [curl:set on page 178](#)
- [curl:single on page 179](#)
- [cURL Examples on page 179](#)

### Understanding the cURL Extension Library

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

To incorporate functions from the cURL extension library into SLAX scripts, include the namespace statement for that library in the script.

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

Call the cURL extension functions using the **curl** prefix and the function name. cURL operations are directed using a set of elements passed to the cURL extension functions.

[Table 17 on page 174](#) lists the supported operations in the cURL extension library and includes a description of each function. [Table 18 on page 175](#) and [Table 19 on page 176](#) list the supported elements and include the syntax and a description of each element. [Table 18 on page 175](#) lists elements used for web services operations, and [Table 19 on page 176](#) lists the elements used for e-mail operations.

Table 17: Functions in the cURL Extension Library

Function	Description
curl:close	Close an open connection. Further operations cannot be performed over the connection. See " <a href="#">curl:close</a> " on page 177.
curl:open	Open a connection to a remote server, allowing multiple operations over a single connection. See " <a href="#">curl:open</a> " on page 177.
curl:perform	Perform simple transfers using a persistent connection handle provided by curl:open. See " <a href="#">curl:perform</a> " on page 177.

Table 17: Functions in the cURL Extension Library (*continued*)

Function	Description
curl:set	Record a set of parameters that persists for the lifespan of a connection. See <a href="#">"curl:set" on page 178</a> .
curl:single	Perform transfer operations without using a persistent connection. See <a href="#">"curl:single" on page 179</a> .

Table 18: Web Services Elements in the cURL Extension Library

Element	Description	Syntax
<content-type>	Provide the MIME type for the transfer payload.	<content-type> "mime/type";
<fail-on-error>	Indicate that the transfer should fail if any errors, including insignificant ones, are detected.	<fail-on-error>;
<format>	Specify the expected format of returned results, allowing the cURL extension to automatically make the content available in the native format. Formats include "html", "text", and "xml".	<format> "xml";
<header>	Provide additional header fields for the request.	<header name="name"> "value";
<insecure>	Indicate a willingness to tolerate insecure communications operations. Specifically, allow SSL Certs without checking the common name.	<insecure>;
<method>	Set the method used to transfer data. This controls the HTTP request type, as well as triggering other transfer mechanisms.  Acceptable method names include "get", "post", "delete", "head", "email", "put", and "upload". The "get" method is the default.	<method> "get";
<param>	Provide additional parameter values for the request. These parameters are typically encoded into the URL.	<param name="x"> "y";
<password>	Set the user's password for the transfer.	<password> "password";
<secure>	Request the use of the secure version of a protocol, including HTTPS and FTPS.	<secure>;

Table 18: Web Services Elements in the cURL Extension Library (*continued*)

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

Table 19: E-mail Elements in the cURL Extension Library

Element	Description	Syntax
<cc>	Set the "Cc" address for e-mail (SMTP) requests. For multiple addresses, use multiple <cc> elements.	<cc> "cc-user@email.example.com";
<contents>	Specify the contents to be transferred.	<contents> "multi-\nline\ncontents\n";
<from>	Set the "From" address for e-mail (SMTP) requests.	<from> "source-user@email.example.com";
<header>	Provide additional header fields for the request.	<header name="name"> "value";
<local>	Set the local hostname for e-mail (SMTP) requests.	<local> "local host name";
<server>	Set the outgoing SMTP server name. Currently, MX records are not processed.	<server> "email-server.example.com";
<subject>	Set the "Subject" field for e-mail (SMTP) requests.	<subject> "email subject string";
<to>	Set the "To" address for e-mail (SMTP) requests. For multiple addresses, use multiple <to> elements.	<to> "to-user@email.examplecom";
<verbose>	Request detailed debug information about the operations and communication of the cURL transfer.	<verbose>;

The libcurl elements closely mimic the options used by the native C libcurl API in libcurl's `curl_easy_setopt()` function. Once the options are set, a call to `curl_easy_perform()`

performs the requested transfer. For more information about the `curl_easy_setopt()` function, see [http://curl.haxx.se/libcurl/c/curl\\_easy\\_setopt.html](http://curl.haxx.se/libcurl/c/curl_easy_setopt.html).

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

These elements can be used in three ways:

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

---

### `curl:close`

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

The syntax is:

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

The argument is the connection handle to close.

---

### `curl:open`

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

The syntax is:

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

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

---

### `curl:perform`

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

The syntax is:

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

The arguments are the connection handle and a set of option elements. Supported cURL extension library elements are defined in [Table 18 on page 175](#) and [Table 19 on page 176](#).

The returned object is an XML hierarchy containing the results of the transfer.

[Table 20 on page 178](#) lists the possible elements in the reply, and [Table 21 on page 178](#) lists the possible elements contained within the **<header>** element.

**Table 20: curl:perform Reply Elements**

Element	Contents
<b>&lt;curl-success&gt;</b>	Empty element which indicates success
<b>&lt;data&gt;</b>	Parsed data
<b>&lt;error&gt;</b>	Error message text, if any
<b>&lt;header&gt;</b>	Parsed header fields
<b>&lt;raw-data&gt;</b>	Raw data from the reply
<b>&lt;raw-headers&gt;</b>	Raw header fields from the reply
<b>&lt;url&gt;</b>	Requested URL

**Table 21: curl:perform <header> Elements**

Element	Contents
<b>&lt;code&gt;</b>	HTTP reply code
<b>&lt;field&gt;</b>	HTTP reply field (with @name and value)
<b>&lt;message&gt;</b>	HTTP reply message
<b>&lt;version&gt;</b>	HTTP reply version string

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

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

### **curl:set**

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



The syntax is:

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

The arguments are the connection handle and a set of option elements. Supported cURL extension library elements are defined in [Table 18 on page 175](#) and [Table 19 on page 176](#).

### **curl:single**

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

The syntax is:

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

The returned object is identical in structure to the one returned by **curl:perform()**. Refer to ["curl:perform" on page 177](#) for additional information.

### **cURL Examples**

The following examples show SLAX version 1.1 scripts that use the cURL extension library functions to perform operations.

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

```
version 1.1;

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

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

match / {
  <op-script-results> {
    var $options = {
      <header name="client"> "slaxproc";
      <param name="smokey"> "bandit";
    }
    var $results = curl:single($url, $options);
    message "completed: " _ $results/headers/message;
    <curl> {
      copy-of $results;
    }
  }
}
```

The following SLAX script takes a username and password and uses the Google login services to translate them into an "Authorization" string:

```
version 1.1;

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

param $url = "https://www.google.com/accounts/ClientLogin";
param $username;
param $password;
```

```
var $auth-params := {
  <url> $url;
  <method> "post";
  <insecure>;
  <param name="Email"> $username;
  <param name="Passwd"> $password;
  <param name="accountType"> "GOOGLE";
  <param name="service"> "wise";
  <param name="source"> "test-app";
}

match / {
  var $curl = curl:open();
  var $auth-cred = curl:perform($curl, $auth-params);

  <options> {
    for-each(slx:break-lines( $auth-cred/raw-data )) {
      if(starts-with(., "Auth")) {
        <header name="GData-Version"> "3.0";
        <header name="Authorization"> "GoogleLogin " _ .;
      }
    }
  }
  expr curl:close($curl);
}
```

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

```
version 1.1;

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

param $server;

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

libslax xutil Extension Library

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

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

Call the xutil extension functions using the **xutil** prefix and the function name. [Table 22 on page 181](#) lists the functions available in the xutil extension library and which are supported in SLAX 1.1 scripts.

Table 22: Functions in the xutil Extension Library

Function and Arguments	Description
xutil:max-call-depth(number)	Changes the limit on the depth of recursive calls. The default limit is 3,000.
xutil:string-to-xml(string+)	Return a node set of the concatenated string arguments.
xutil:xml-to-string(node-set+)	Return a string representation of the XML hierarchies provided as arguments.

Related Documentation

- [libslax Library and Extension Libraries Overview on page 110](#)
- [libslax Distribution Overview on page 109](#)
- [Downloading and Installing the libslax Distribution on page 171](#)
- [Understanding the SLAX Processor \(slaxproc\) on page 111](#)
- [Using the SLAX Processor \(slaxproc\) on page 187](#)
- [SLAX Debugger, Profiler, and callflow on page 181](#)

SLAX Debugger, Profiler, and callflow

- [SLAX Debugger, Profiler, and callflow Overview on page 181](#)
- [Using the SLAX Debugger, Profiler, and callflow on page 183](#)

SLAX Debugger, Profiler, and callflow Overview

The Junos OS command-line interface (CLI) and the libslax distribution include the SLAX debugger (sdb), which is used to trace the execution of SLAX scripts. The SLAX debugger enables you to step through script execution, pause script execution at defined breakpoints, and review the value of script variables at any point.

The SLAX debugger operation and command syntax resemble that of the GNU Project Debugger (GDB). Many of the sdb commands follow their GDB counterparts, to the extent possible. [Table 23 on page 182](#) lists the SLAX debugger commands and a brief description of each command.

The SLAX debugger includes a profiler that can report information about the activity and performance of a script. The profiler, which is automatically enabled when you start the debugger, tracks script execution until the script terminates. At any point, profiling information can be displayed or cleared, and the profiler can be temporarily disabled or enabled. The SLAX debugger **callflow** command enables printing of informational data when you enter or exit levels of the script.

**Table 23: SLAX Debugger Commands**

Command	Description
<code>break [loc]</code>	Add a breakpoint to the script at the current line of execution. Optionally specify <code>[file:]line</code> or a template name to create a breakpoint at that position.
<code>callflow [on   off]</code>	Enable or disable callflow tracing. You can explicitly specify the <b>on</b> or <b>off</b> value. Omitting the value toggles callflow on and off.
<code>continue [loc]</code>	Continue running the script until it reaches the next breakpoint. If there are no defined breakpoints, the script runs in its entirety. Optionally, specify <code>[file:]line</code> or a template name. When you include the optional argument, script execution continues until it reaches either a breakpoint or the specified line number or template name, whichever comes first.
<code>delete [num]</code>	Delete one or all breakpoints. Breakpoints are numbered sequentially as they are created. Omit the optional argument to delete all breakpoints. Include the breakpoint number as an argument to delete only the specified breakpoint. View currently active breakpoints with the <b>info</b> command.
<code>finish</code>	Finish executing the current template.
<code>help</code>	Display the help message.
<code>info [breakpoints   profile   profile brief]</code>	Display information about the current script. The default command lists all breakpoints in the script. Optionally specify the <b>profile</b> or <b>profile brief</b> arguments to display profiling information.
<code>list [loc]</code>	List the contents of the current script. Optionally specify <code>[file:]line</code> or a template name from which point the debugger lists partial script contents. The output includes the filename, line number, and code.
<code>next</code>	Execute the next instruction, stepping over any function or template calls.
<code>over</code>	Execute the next instruction, stepping over any function or template calls or instruction hierarchies.
<code>print &lt;xpath&gt;</code>	Print the value of the XPath expression.
<code>profile [clear   on   off   report   report brief]</code>	Enable or disable the profiler. The profiler is enabled by default. Include the <b>clear</b> option to clear profiling information. Include the <b>report</b> or <b>report brief</b> option to display profiling information for the current script.
<code>quit</code>	Exit debugging mode.

Table 23: SLAX Debugger Commands (*continued*)

Command	Description
reload	Reload the script.
run	Restart script execution from the beginning of the script.
step	Execute the next instruction, stepping into any function or template calls or instruction hierarchies.
where	Show the backtrace of template calls.

## Using the SLAX Debugger, Profiler, and callflow

- [Invoking the SLAX Debugger on page 183](#)
- [Using the SLAX Debugger \(sdb\) on page 184](#)
- [Using the SLAX Profiler on page 185](#)
- [Using callflow on page 187](#)

### Invoking the SLAX Debugger

Both the Junos OS CLI and the SLAX processor in the libslax distribution include the SLAX debugger (sdb), which is used to trace the execution of SLAX scripts.

When you invoke the SLAX debugger, the command-line prompt changes to (sdb) to indicate that you are in debugging mode. For example:

```
sdb: The SLAX Debugger (version )
Type 'help' for help
(sdb)
```

When using the SLAX debugger from the Junos OS CLI, you can only use the debugger with op scripts that are enabled in the configuration. To invoke the SLAX debugger from the CLI on a device running Junos OS, issue the **op invoke-debugger cli** operational mode command, include the op script name, and optionally include any necessary script arguments.

```
user@host>op invoke-debugger cli script <argument-name argument-value>
```

The following example invokes the SLAX debugger for the op script **ge-interfaces.slax**, which has two parameters, **interface** and **protocol**. Values are supplied for both arguments.

```
user@host> op invoke-debugger cli ge-interfaces interface ge-0/2/0.0 protocol inet
sdb: The SLAX Debugger (version )
Type 'help' for help
(sdb)
```

To invoke the SLAX debugger when using the SLAX processor, issue the **slaxproc** command with the **--debug** or **-d** option. Specify the script file and any input or output files. If no input file is required, use the **-E** option to indicate an empty input document. If the **-i** or **--input** argument has the value **"-"**, or if you do not include the input option or an input file, standard input is used. When using standard input, press Ctrl+d to signal the end-of-file. The general syntax is:

```
$ slaxproc --debug [options] [script] [files]
```

The following example invokes the SLAX debugger for the script **script1.slax** with an empty input document and an output file **script1-output.xml**

```
$ slaxproc --debug -n script1.slax -o script1-output.xml -E
sdb: The SLAX Debugger (version )
Type 'help' for help
(sdb)
```

### Using the SLAX Debugger (sdb)

---

To view the SLAX debugger help message, issue the **help** command at the (sdb) prompt. To display the help message for a single command, issue **help command**, where *command* is the sdb command for which you want more information. For example:

```
(sdb) help break
break [loc]  Add a breakpoint at [file:]line or template
```

The process for debugging a script varies depending on the script. A generic outline is presented here:

1. Enter debugging mode.
2. Insert breakpoints in the script using the **break** command.

During execution, the debugger pauses at defined breakpoints.

The breakpoint location can be the name of a template or a line number in the current script, or the filename and a line number separated by a colon. If you do not include an argument, a breakpoint is created at the current line of execution. Breakpoints are numbered sequentially as you create them. To view a list of breakpoints, issue the **info breakpoints** command. To delete a breakpoint, issue the **delete num** command, and specify the breakpoint number. To delete all breakpoints, issue the **delete** command with no argument.

The following example creates three breakpoints, the first at line 7, the second at line 25, and the third at the template named "three":

```
(sdb) break 7
Breakpoint 1 at file script1.slax, line 7
(sdb) break 25
Breakpoint 2 at file script1.slax, line 25
(sdb) break three
Breakpoint 3 at file script1.slax, line 51
(sdb) info breakpoints
List of breakpoints:
#1 [global] at script1.slax:7
#2 template two at script1.slax:25
#3 template three at script1.slax:51
```

- Increment script execution by issuing the **continue**, **finish**, **next**, **over**, and **step** commands at the debugger prompt.

For example:

```
(sdb) next
Reached breakpoint 1, at script1.slax:7
script1.slax:3: var $byte = "10011001";
```

- Review the value of variables as the program executes to ensure that they have the expected value.

```
print xpath-expression
```

- To reload the script contents at any point and restart script execution from the beginning, issue the **reload** command.

```
(sdb) reload
The script being debugged has been started already.
Reload and restart it from the beginning? (y or n) y
Reloading script...
Reloading complete.
```

### Using the SLAX Profiler

The SLAX debugger includes a profiler that can report information about the activity and performance of a script. The profiler, which is automatically enabled when you start the debugger, tracks script execution until the script terminates. At any point, profiling information can be displayed or cleared, and the profiler can be temporarily disabled or enabled.

To access the profiler, issue the **profile** command at the SLAX debugger prompt, (sdb), and include any options. The profile command syntax is:

```
(sdb) profile [options]
```

[Table 24 on page 185](#) lists the profile command options. Issuing the **profile** command with no additional options toggles the profiler on and off.

```
(sdb) profile
Disabling profiler
(sdb)
```

You can access the profiler help by issuing the **help profile** command at the (sdb) prompt.

**Table 24: Profile command options**

Option	Description
clear	Clear profiling information
off	Disable profiling
on	Enable profiling
report [brief]	Report profiling information

To enable the profiler and print a report:

1. Enter debugging mode. The profiler is enabled by default.
2. Step through script execution, or execute a script in its entirety.

```
(sdb) run
<?xml version="1.0"?>
<message>Down rev PIC in Fruvenator, Fru-Master 3000</message>
Script exited normally.
```

3. At any point during script execution, display profiling information.

The **brief** option instructs sdb to avoid showing lines that were not hit, since there is no valid information. If you omit the **brief** option, dashes are displayed.

```
(sdb) profile report brief
```

The following sample output shows a profile report with and without the **brief** option. The source code data in the example is truncated for display purposes.

```
(sdb) profile report
```

Line	Hits	User	U/Hit	System	S/Hit	Source
1	-	-	-	-	-	version 1.0;
2	-	-	-	-	-	
3	2	4	2.00	8	4.00	match / {
4	1	25	25.00	13	13.00	var ....
5	-	-	-	-	-	
6	-	-	-	-	-	for-each....
7	1	45	45.00	10	10.00	..
8	1	12	12.00	5	5.00	<message>
9	1	45	45.00	15	15.00	....
10	-	-	-	-	-	}
11	-	-	-	-	-	- }
Total	6	131		51	Total	

```
(sdb) pro rep b
```

Line	Hits	User	U/Hit	System	S/Hit	Source
3	2	4	2.00	8	4.00	match / {
4	1	25	25.00	13	13.00	var ....
7	1	45	45.00	10	10.00	....
8	1	12	12.00	5	5.00	<message>
9	1	45	45.00	15	15.00	....
Total	6	131		51	Total	

The profile report includes the following information:

- **Line**—Line number in the source file.
- **Hits**—Number of times this line was executed.
- **User**—Number of microseconds of "user" time spent processing this line.
- **U/Hit**—Average number of microseconds of "user" time per hit.
- **System**—Number of microseconds of "system" time spent processing this line.
- **S/Hit**—Average number of microseconds of "system" time per hit.
- **Source**—Source code line.



This information not only shows how much time is spent during code execution, but can also show which lines are being executed, which can help debug scripts where the execution does not match expectations.

### Using callflow

---

The SLAX debugger **callflow** command enables printing of informational data when you enter or exit levels of the script.

To enable callflow and view callflow data for a script:

1. Enter debugging mode.
2. Issue the **callflow** command at the SLAX debugger prompt, (sdb).

```
(sdb) callflow
Enabling callflow
```

3. Step through script execution, or execute a script in its entirety.

Callflow prints information as it enters and exits different levels of the script. Each output line references the instruction, filename, and line number of the frame.

```
(sdb) run
callflow: 0: enter <xsl:template> in match / at script3.slax:5
callflow: 1: enter <xsl:template> in template one at script3.slax:14
callflow: 2: enter <xsl:template> in template two at script3.slax:20
callflow: 3: enter <xsl:call-template> at script3.slax:22
...
<?xml version="1.0"?>
<message>Down rev PIC in Fruvenator, Fru-Master 3000</message>
Script exited normally.
```

#### Related Documentation

- [op invoke-debugger cli on page 276](#)
- [Understanding the SLAX Processor \(slaxproc\) on page 111](#)
- [Using the SLAX Processor \(slaxproc\) on page 187](#)
- [libslax Distribution Overview on page 109](#)

## Using the SLAX Processor (slaxproc)

---

The SLAX processor (slaxproc) is a command-line tool that can validate SLAX script syntax, convert between SLAX and XSLT formats, and format or run SLAX scripts. The slaxproc mode options define what function the processor performs. The following sections outline each of the modes:

- [Validating SLAX Script Syntax on page 188](#)
- [Converting Scripts Between XSLT and SLAX Formats on page 188](#)
- [Running SLAX Scripts on page 190](#)
- [Formatting SLAX Scripts on page 191](#)

## Validating SLAX Script Syntax

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

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

```
$ slaxproc --check script1.slax
```

OR

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

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

## Converting Scripts Between XSLT and SLAX Formats

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

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

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

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

OR

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

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

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

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

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

OR

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

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

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

The SLAX processor returns the SLAX formatting for the hierarchy.

```
<policy-options> {
  <policy-statement> {
    <name> "export-policy";
    <term> {
      <name> "term1";
      <from> {
        <route-filter> {
          <address> "10.0.4.4/30";
          <prefix-length-range> "/30-/30";
        }
      }
    }
  }
}
```

```
    }  
    <then> {  
      <accept>;  
    }  
  }  
}
```

## Running SLAX Scripts

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

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

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

The syntax for executing a script is:

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

or

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

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

1. Create a script using your favorite editor.
2. (Optional) Check the script syntax by invoking the processor with the `--check` or `-c` mode option, and fix any indicated errors.

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

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

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

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

```
$ slaxproc script1.slax input.xml output.xml
```

```
$ slaxproc -n script1.slax -i input.xml -o output.xml
```

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

```
$ slaxproc -E script1.slax output.xml
```

```
$ slaxproc -n script1.slax -o output.xml -E
```

- To execute a script and use standard input as the input document, issue the **slaxproc** command with no input file argument. At the prompt, enter the input and press Ctrl+d to signal the end-of-file. For example:

```
$ slaxproc -n script1.slax -o output.xml
<user input>
[Ctrl+d]
```

## Formatting SLAX Scripts

The SLAX processor provides the option to format a script to correct the indentation and spacing to the preferred style. When you format a script, you have the option to reference the file arguments positionally or use the command-line file options, **--input** or **-i** and **--output** or **-o**, to specify the unformatted input file and the formatted output file, respectively. If you use the command-line file options, the files can be referenced in any order on the command line.

To format a SLAX script, issue the **slaxproc** command with the **--format** or **-F** mode option. To reference the files positionally, specify the unformatted SLAX script as the first argument and the desired output path and filename of the formatted SLAX script as the second argument. To reference the files using command-line file options, include the file options in any order. For example:

```
$ slaxproc --format script1.slax script1-format.slax
```

OR

```
$ slaxproc -F -i script1.slax -o script1-format.slax
```

Given the following unformatted SLAX script as input:

```
version 1.1;

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

match / {
var $number = -14560302.5;
```

```
    expr format-number($number, "###,###.00", "default-format");  
}
```

the SLAX processor outputs the following formatted SLAX script:

```
version 1.1;  
  
decimal-format default-format {  
    decimal-separator ".";  
    digit "#";  
    grouping-separator ",";  
    infinity "Infinity";  
    minus-sign "-";  
    pattern-separator ",";  
    percent "%";  
    per-mille " 30";  
    zero-digit "0";  
    nan "NaN";  
}  
  
match / {  
    var $number = -14560302.5;  
  
    expr format-number($number, "###,###.00", "default-format");  
}
```

**Related  
Documentation**

- [libslax Distribution Overview on page 109](#)
- [libslax Library and Extension Libraries Overview on page 110](#)
- [Understanding the SLAX Processor \(slaxproc\) on page 111](#)

## CHAPTER 15

# SLAX Statements

- [append on page 194](#)
- [apply-imports on page 195](#)
- [apply-templates on page 195](#)
- [attribute on page 196](#)
- [attribute-set on page 197](#)
- [call on page 199](#)
- [copy-node on page 200](#)
- [copy-of on page 201](#)
- [decimal-format on page 201](#)
- [element on page 203](#)
- [else on page 203](#)
- [else if on page 204](#)
- [expr on page 205](#)
- [fallback on page 206](#)
- [for on page 207](#)
- [for-each on page 208](#)
- [function on page 210](#)
- [if on page 211](#)
- [import on page 212](#)
- [key on page 213](#)
- [match on page 215](#)
- [message on page 216](#)
- [mode on page 216](#)
- [mvar on page 217](#)
- [number on page 218](#)
- [output-method on page 222](#)
- [param on page 225](#)
- [preserve-space on page 226](#)

- [priority on page 226](#)
- [processing-instruction on page 227](#)
- [result on page 229](#)
- [set on page 230](#)
- [sort on page 230](#)
- [strip-space on page 232](#)
- [template on page 232](#)
- [terminate on page 234](#)
- [trace on page 234](#)
- [uexpr on page 235](#)
- [use-attribute-sets on page 236](#)
- [var on page 237](#)
- [version on page 237](#)
- [while on page 238](#)
- [with on page 239](#)

---

## append

---

<b>Syntax</b>	<code>append <i>name</i> += <i>value</i>;</code>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	Append a value to the node set contained in a mutable variable. The variable must be defined using the <b>mvar</b> statement.
<b>Attributes</b>	<i>name</i> —Name of the mutable variable.  <i>value</i> —Value to append to the node set.
<b>SLAX Example</b>	<p>The following snippet appends the <code>&lt;item&gt;</code> element and <code>&lt;name&gt;</code> and <code>&lt;size&gt;</code> child elements to the node set contained in the mutable variable <b>block</b>:</p> <pre>mvar \$block; set \$block = &lt;block&gt; "item list";  for \$item (list) {   append \$block += &lt;item&gt; {     &lt;name&gt; \$item/name;     &lt;size&gt; \$item/size;   } }</pre>
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">SLAX Variables Overview on page 97</a></li><li>• <a href="#">mvar on page 217</a></li><li>• <a href="#">set on page 230</a></li></ul>



## apply-imports

<b>Syntax</b>	<code>apply-imports;</code>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	<p>Apply a template rule from an external file or style sheet. By default, template rules in the main script have precedence over equivalent imported template rules. Use this statement to process the context node using the imported match template rule from the external source.</p> <p>The <b>apply-imports</b> statement mimics the <code>&lt;xsl:apply-imports&gt;</code> element, allowing the script to invoke any imported templates.</p>
<b>SLAX Example</b>	<p>In the example, the main script imports the file <b>route-rules.slax</b>. The <b>apply-imports</b> statement invokes the imported template rule for <b>&lt;route&gt;</b> elements.</p> <pre> version 1.1; import "route-rules.slax";  match route {   &lt;routes&gt; {     apply-imports;   } }</pre> <p>The imported file contains a template rule for <b>&lt;route&gt;</b> elements.</p> <pre> /* route-rules.slax */ version 1.1;  match route {   &lt;new&gt; {     apply-templates *[@changed];   } }</pre>
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">import on page 212</a></li> </ul>

## apply-templates

<b>Syntax</b>	<code>apply-templates <i>expression</i>;</code>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	<p>Apply one or more templates, according to the value of the node-set expression. If a node-set expression is not specified, the script recursively processes all child nodes of the current node. If a node-set expression is specified, the processor only applies templates to the child elements that match the node-set expression. The <b>template</b> statement dictates which elements are transformed according to which template. The</p>

templates that are applied are passed the parameters specified by the **with** statement within the **apply-templates** statement block.

<b>Attributes</b>	<i>expression</i> —(Optional) Selects the nodes to which the processor applies templates. By default, the processor applies templates to the child nodes of the current node.
<b>SLAX Example</b>	<pre>match configuration {   apply-templates system/host-name; }</pre>
<b>XSLT Equivalent</b>	<pre>&lt;xsl:template match="configuration"&gt;   &lt;xsl:apply-templates select="system/host-name"/&gt; &lt;/xsl:template&gt;</pre>
<b>Usage Examples</b>	See <i>Example: Adding a Final then accept Term to a Firewall</i> and <i>Example: Preventing Import of the Full Routing Table</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">SLAX Templates Overview on page 88</a></li><li>• <a href="#">call on page 199</a></li><li>• <a href="#">match on page 215</a></li><li>• <a href="#">mode on page 216</a></li><li>• <a href="#">priority on page 226</a></li><li>• <a href="#">template on page 232</a></li><li>• <a href="#">with on page 239</a></li></ul>

---

## attribute

---

<b>Syntax</b>	<pre>attribute <i>attribute-name</i> {   <i>attribute-value</i>; }</pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	Create an attribute with the given name. The attribute value is defined by a block of statements, which must be placed inside a set of braces.
<b>Attributes</b>	<p><i>attribute-name</i>—Name of the attribute, which can be an XPath expression or a string. Enclose string arguments in quotes.</p> <p><i>attribute-value</i>—A block of statements enclosed in curly braces that defines the attribute value.</p>

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

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

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

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

- Related Documentation**
- [SLAX Elements and Element Attributes Overview on page 85](#)
  - [attribute-set on page 197](#)
  - [element on page 203](#)
  - [use-attribute-sets on page 236](#)

## attribute-set

**Syntax**

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

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

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

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

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

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

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

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

```
version 1.1;

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

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

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

#### XSLT Equivalent

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:variable name="cellpadding" select="0"/>
  <xsl:variable name="cellspacing" select="10"/>
  <xsl:attribute-set name="table-attributes">
    <xsl:attribute name="order">
      <xsl:text>0</xsl:text>
    </xsl:attribute>
    <xsl:attribute name="cellpadding">
      <xsl:value-of select="$cellpadding"/>
    </xsl:attribute>
    <xsl:attribute name="cellspacing">
      <xsl:value-of select="$cellspacing"/>
    </xsl:attribute>
  </xsl:attribute-set>
  <xsl:attribute-set name="table-attributes-ext" use-attribute-sets="table-attributes">

    <xsl:attribute name="border">
      <xsl:text>0</xsl:text>
    </xsl:attribute>
  </xsl:attribute-set>
  <xsl:template match="/">
    <table use-attribute-sets="table-attributes-ext"/>
```

```
</xsl:template>
</xsl:stylesheet>
```

- Related Documentation**
- [SLAX Elements and Element Attributes Overview on page 85](#)
  - [attribute on page 196](#)
  - [element on page 203](#)
  - [use-attribute-sets on page 236](#)

## call

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

**Release Information**    Statement introduced in version 1.0 of the SLAX language.

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

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

**Attributes**            *template-name*—Specifies the name of the template to call.

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

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

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

**Usage Examples** See *Example: Requiring and Restricting Configuration Statements*, *Example: Imposing a Minimum MTU Setting*, and *Example: Automatically Configuring Logical Interfaces and IP Addresses*.

- Related Documentation**
- [SLAX Templates Overview on page 88](#)
  - [apply-templates on page 195](#)
  - [match on page 215](#)
  - [mode on page 216](#)
  - [priority on page 226](#)
  - [template on page 232](#)
  - [with on page 239](#)

---

## copy-node

---

**Syntax**

```
copy-node;

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

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

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

**SLAX Example**

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

**XSLT Equivalent**

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

- Related Documentation**
- [copy-of on page 201](#)
  - [xsl:copy-of on page 250](#)

## copy-of

<b>Syntax</b>	<code>copy-of expression;</code>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	Copy the specified node including namespace nodes, child nodes, and attributes of that node. The argument is an XPath expression that specifies which nodes to copy.
<b>Attributes</b>	<i>expression</i> —XPath expression that specifies which nodes to copy.
<b>SLAX Example</b>	<code>copy-of configuration/protocols/bgp;</code>
<b>XSLT Equivalent</b>	<code>&lt;xsl:copy-of select="configuration/protocols/bgp"/&gt;</code>
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">copy-node on page 200</a></li> <li>• <a href="#">xsl:copy-of on page 250</a></li> </ul>

## decimal-format

<b>Syntax</b>	<pre>decimal-format <i>format-name</i> {     decimal-separator <i>character</i>;     digit <i>character</i> ;     grouping-separator <i>character</i>;     infinity <i>string</i>;     minus-sign <i>character</i>;     nan <i>string</i>;     pattern-separator <i>character</i>;     percent <i>character</i>;     per-mille <i>character</i>;     zero-digit <i>character</i>; }</pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	Define formatting parameters for use by the <b>format-number()</b> XPath function. The <b>decimal-format</b> statement must be defined as a top-level statement in the script.
<b>Attributes</b>	<p><i>decimal-format format-name</i>—Decimal-format identifier, which is passed as the third argument to the <b>format-number()</b> XPath function.</p> <p><i>decimal-separator character</i>—Character used as the decimal sign. The default is the period (.).</p> <p><i>digit character</i>—Character used to represent a digit in a pattern. The default is the number sign (#).</p> <p><i>grouping-separator character</i>—Character used as the digit group separator or the thousands separator. The default is the comma (,).</p>

*infinity string*—String used to represent infinity. The default is "Infinity".

*minus-sign character*—Character used as the minus sign. The default is the hyphen (-).

*nan string*—String used to represent NaN. The default is "NaN".

*pattern-separator character*—Character used to separate patterns. The first pattern is used for positive numbers, and the second pattern is used for negative numbers. The default is the semicolon (;).

*percent character*—Character used as the percent sign. The default is the percent character (%).

*per-mille character*—Character used as a per mille sign. The default is the Unicode per mille sign (\x2030 or ‰).

*zero-digit character*—Character used as zero. The default is the number zero (0).

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

```
version 1.1;

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

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

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

**XSLT Equivalent**

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

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

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



```
</xsl:template>
</xsl:stylesheet>
```

**Related Documentation** • [output-method on page 222](#)

## element

<b>Syntax</b>	<pre>element <i>name</i> {     /* element contents */ }</pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	Create an element node with the given name. The element name can be an XPath expression or a string. The element contents must be placed inside curly braces.
<b>Attributes</b>	<i>name</i> —Name of the element, which can be an XPath expression or a string. Enclose string arguments in quotes.
<b>SLAX Example</b>	The following sample code uses the value of the <b>name</b> node (rather than the literal string "name") to create an XML element, whose contents are an empty element with a name of "from-" concatenated with the value of the address node. Node values are selected from the current context.

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

**Related Documentation** • [SLAX Elements and Element Attributes Overview on page 85](#)

## else

<b>Syntax</b>	<pre>if (<i>expression</i>) {     /* code */ } else {     /* code */ }</pre>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	Include a default set of instructions that are processed if the preceding <b>if</b> and <b>else if</b> statements evaluate to <b>FALSE</b> .
<b>SLAX Example</b>	<pre>if (starts-with(name, "fe-")) {     if (mtu &lt; 1500) {         /* Select the Fast Ethernet interfaces with low MTUs */     } }</pre>

	<pre>    }   }   else {     if (mtu &gt; 8096) {       /* Select the non-Fast Ethernet interfaces with high MTUs */     }   } }</pre>
XSLT Equivalent	<pre>&lt;xsl:choose&gt;   &lt;xsl:when select="starts-with(name, 'fe-')"&gt;     &lt;xsl:if test="mtu &lt; 1500"&gt;       &lt;!-- Select with Fast Ethernet interfaces with low MTUs --&gt;     &lt;/xsl:if&gt;   &lt;/xsl:when&gt;   &lt;xsl:otherwise&gt;     &lt;xsl:if test="mtu &gt; 8096"&gt;       &lt;!-- Select the non-Fast Ethernet interfaces with high MTUs --&gt;     &lt;/xsl:if&gt;   &lt;/xsl:otherwise&gt; &lt;/xsl:choose&gt;</pre>
Usage Examples	See <i>Example: Configuring Dual Routing Engines</i> and <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .
Related Documentation	<ul style="list-style-type: none"><li>• <a href="#">SLAX Statements Overview on page 100</a></li><li>• <a href="#">else if on page 204</a></li><li>• <a href="#">for-each on page 208</a></li><li>• <a href="#">if on page 211</a></li></ul>

---

## else if

Syntax	<pre>if (<i>expression</i>) {   /* code */ } else if (<i>expression</i>) {   /* code */ }</pre>
Release Information	Statement introduced in version 1.0 of the SLAX language.
Description	Include instructions that are processed if the expression defined in the preceding <b>if</b> statement evaluates to <b>FALSE</b> and the expression defined in the <b>else if</b> statement evaluates to <b>TRUE</b> . Multiple <b>else if</b> statements can be included, but the processor only executes the instructions contained in the first <b>else if</b> statement whose expression evaluates to <b>TRUE</b> . All subsequent <b>else if</b> statements are ignored.
SLAX Example	<pre>var \$description2 = {   if (description) {     expr description;   }   else if (../description) {</pre>

	<pre>         expr ../description;     }     else {         expr "no description found";     } } </pre>
<b>XSLT Equivalent</b>	<pre> &lt;xsl:variable name="description2"&gt;   &lt;xsl:choose&gt;     &lt;xsl:when test="description"&gt;       &lt;xsl:value-of select="description"/&gt;     &lt;/xsl:when&gt;     &lt;xsl:when test="../description"&gt;       &lt;xsl:value-of select="../description"/&gt;     &lt;/xsl:when&gt;     &lt;xsl:otherwise&gt;unknown&lt;/xsl:otherwise&gt;   &lt;/xsl:choose&gt; &lt;/xsl:variable&gt; </pre>
<b>Usage Examples</b>	See <i>Example: Configuring Dual Routing Engines</i> and <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">SLAX Statements Overview on page 100</a></li> <li>• <a href="#">else on page 203</a></li> <li>• <a href="#">for-each on page 208</a></li> <li>• <a href="#">if on page 211</a></li> </ul>

## expr

<b>Syntax</b>	<code>expr expression;</code>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	<p>Generate the string value of an XPath expression and add it to the result tree. The XPath expression might consist of a function call, a location path, a literal number, or a string. SLAX-specific operators are permitted. This statement cannot be used at the top-level of a script. It can only appear within a code block. By default, characters such as "&lt;", "&gt;", and "&amp;" are escaped into proper XML as "&amp;lt;", "&amp;gt;", and "&amp;amp;", respectively.</p> <p>The <b>expr</b> statement is most commonly used to invoke functions that return no results, for conditional variable assignment, and to return text content from a template.</p>
<b>Attributes</b>	<i>expression</i> —XPath expression to evaluate. The resulting string is added to the result tree.
<b>SLAX Example</b>	<pre> expr "Test: "; expr substring-before(name, "."); expr status; expr jcs:output("Test"); </pre>

<b>XSLT Equivalent</b>	<pre>&lt;xsl:text&gt;Test: &lt;/xsl:text&gt; &lt;xsl:value-of select="substring-before(name, '.')" /&gt; &lt;xsl:value-of select="status" /&gt; &lt;xsl:value-of select="jcs:output('Test')" /&gt;</pre>
<b>Usage Examples</b>	<ul style="list-style-type: none"><li>• <i>Example: Adding a Final then accept Term to a Firewall</i></li><li>• <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i></li></ul>
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">XPath Expressions Overview for SLAX on page 87</a></li><li>• <a href="#">message on page 216</a></li><li>• <a href="#">terminate on page 234</a></li><li>• <a href="#">trace on page 234</a></li><li>• <a href="#">uexpr on page 235</a></li></ul>

---

## fallback

---

<b>Syntax</b>	<pre>fallback {     /* body to execute if extension function        or element is unavailable */ }</pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	<p>Specify statements to use when an extension function or element is not available in the current implementation. The <b>fallback</b> statement is enclosed within another instruction element to indicate what fallback code should be run if the script processor does not recognize the enclosing instruction element. The script executes the body of the fallback statement to handle this error condition.</p> <p>A script might utilize this statement when it is run in environments that support different extension elements.</p>
<b>SLAX Example</b>	<p>The following example op script declares the namespace binding <b>test</b> with a URI of "test". The code attempts to reference the nonexistent extension element <b>&lt;test:fake&gt;</b>, which is not supported, and the code instead executes the fallback instructions.</p> <pre>version 1.1; ns junos = "http://xml.juniper.net/junos/*/junos"; ns xnm = "http://xml.juniper.net/xnm/1.1/xnm"; ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0"; ns test extension = "test";  match / {   &lt;op-script-results&gt; {     /* Fake extension element */     &lt;test:fake&gt; {       expr slax:output( "&lt;test:fake&gt; exists!" );     }   } }</pre>

```

        fallback {
            expr slax:output( "<test:fake> does not exist." );
        }
    }
}

```

Related •  
Documentation

## for

<b>Syntax</b>	<pre> for <i>name</i> (<i>expression</i>) {     /* code */ }  for <i>name</i> (<i>min</i> ... <i>max</i>) {     /* code */ } </pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	<p>Iterate through an integer set or a node set without changing the context, and execute a block of statements using each member of the integer or node set as the value of the given variable.</p> <p>If the argument is an XPath expression, the variable is assigned each member of the node set selected by the expression in sequence. If the argument is an integer set, the iteration operator (...) generates a sequence of nodes with the value of each integer between the left and right operands. If the left operand is greater than the right operand, the numbers are generated in decreasing order. The variable takes on the value of each integer in sequence. For each iteration, the contents are then evaluated, processed according to the instructions contained in the <b>for</b> code block.</p>
<b>Attributes</b>	<p><i>expression</i>—XPath expression that selects the nodes to be processed.</p> <p><i>max</i>—Integer or variable that defines the end value of the integer sequence. If the end value is less than the start value, the numbers are generated in decreasing order.</p> <p><i>min</i>—Integer or variable that defines the starting value of the integer sequence. If the start value is greater than the end value, the numbers are generated in decreasing order.</p> <p><i>name</i>—Identifier of the <b>for</b> loop variable, which takes on the values of each member of the integer or node set. This variable can be referenced within the <b>for</b> loop code block.</p>

**SLAX Example** In the following example, the **for** loop iterates over the **interfaces** node. The XPath expression selects each **name** node that is a child of the **interface** node and that has a value beginning with the 'ge-' designator. The selection is assigned to the **\$name** variable, which is used within that iteration of the **for** loop code block. The **for** loop outputs a **<name>** element for each selection. The content of each **<name>** element is the interface name currently stored in the **\$name** variable for that iteration. The end result is a list of all Gigabit Ethernet interfaces on the device.

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

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

```
for $int (1 ... 3) {  
  <item> {  
    attribute "item-number" {  
      expr $int;  
    }  
  }  
}  
  
/* Output: <item item-number="1"/><item item-number="2"/><item item-number="3"/>  
*/
```

- Related Documentation**
- [SLAX Statements Overview on page 100](#)
  - [XPath Overview on page 65](#)
  - [for-each on page 208](#)

---

## for-each

---

<b>Syntax</b>	<pre>for-each (<i>expression</i>) {   /* code */ }  /* Syntax added in version 1.1 of the SLAX language.*/ for-each (<i>min</i> ... <i>max</i>) {   /* code */ }</pre>
---------------	--

<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language. Support for iteration operator (...) added in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
----------------------------	--

<b>Description</b>	Include a looping mechanism that repeats script processing for each XML element in the specified node set or each value in the integer set.
--------------------	---

If the argument is an XPath expression, the element nodes are selected by the value of the XPath expression. If the argument is an integer set, the iteration operator (...) generates

a sequence of nodes with the value of each integer between the left and right operands. If the left operand is greater than the right operand, the numbers are generated in decreasing order. For each iteration, the contents are then evaluated, processed according to the instructions contained in the **for-each** code block.

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

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

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

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

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

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

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

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

- Usage Examples**
- *Example: Adding T1 Interfaces to a RIP Group*
  - *Example: Configuring Administrative Groups for LSPs*
  - *Example: Configuring Dual Routing Engines*
  - *Example: Imposing a Minimum MTU Setting*
  - *Example: Limiting the Number of E1 Interfaces*
  - *Example: Requiring and Restricting Configuration Statements*

- Related Documentation**
- [SLAX Statements Overview on page 100](#)
  - [XPath Overview on page 65](#)
  - [for on page 207](#)

- [xsl:for-each on page 251](#)

## function

<b>Syntax</b>	<pre> function <i>function-name</i> (<i>argument-list</i>) {     ...     result <i>return-value</i>; }  function <i>function-name</i> () {     param <i>param-name1</i>;     param <i>param-name2</i>;     param <i>param-name3</i> = <i>default-value</i>;     ...     result <i>return-value</i>; } </pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	<p>Define an extension function that can be used in XPath expressions. The <b>function</b> statement must be defined as a top-level statement in the script using a qualified name for the function identifier. The argument list is a comma-separated list of parameter names, which are positionally assigned based on the function call. Trailing arguments can have default values. Alternatively, you can define function parameters inside the function block using the <b>param</b> statement. The function body is a set of statements, which should include a <b>result</b> statement that defines the return value for the function.</p> <p>If there are fewer arguments in the function invocation than in the definition, the default values are used for any trailing arguments. If there are more arguments in the function invocation than in the definition, the function call generates an error.</p>
<b>Attributes</b>	<p><i>function-name</i>—Specifies the name of the function as a qualified name.</p> <p><i>argument-list</i>—Comma-separated list of parameter names, which are positionally assigned based on the function call. Trailing arguments can have default values.</p> <p><i>return-value</i>—XML element or XPath expression, scalar value, or a set of instructions providing the return value of the function.</p>
<b>SLAX Example</b>	<p>The following example defines the function <b>size</b>, which has three parameters: <b>width</b>, <b>height</b>, and <b>scale</b>. The default value for <b>scale</b> is 1. If the function call argument list does not include the <b>scale</b> argument, the calculation uses the default value of 1 for that argument. The function's return value is the product of the <b>width</b>, <b>height</b>, and <b>scale</b> variables enclosed in a <b>&lt;size&gt;</b> element.</p> <p>In the main match template, the function call uses width and height data selected from each <b>graphic/dimension</b> element in the source XML file. The script evaluates the function, and the <b>copy-of</b> statement emits the return value to the result tree as the contents of the <b>&lt;out&gt;</b> element.</p> <pre> version 1.1; ns my = "http://www.example.com/myfunctions"; </pre>



```

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

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

```

- Related Documentation**
- [SLAX Functions Overview on page 91](#)
  - [param on page 225](#)
  - [result on page 229](#)

## if

**Syntax**

```

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

```

**Release Information** Statement introduced in version 1.0 of the SLAX language.

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

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

**Attributes** *expression*—Specifies the expression to evaluate.

**SLAX Example**

```

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

```

	<pre>    expr ../description;   }   else {     expr "no description found";   } }</pre>
<b>XSLT Equivalent</b>	<pre>&lt;xsl:variable name="description2"&gt;   &lt;xsl:choose&gt;     &lt;xsl:when test="description"&gt;       &lt;xsl:value-of select="description"/&gt;     &lt;/xsl:when&gt;     &lt;xsl:when test="../description"&gt;       &lt;xsl:value-of select="../description"/&gt;     &lt;/xsl:when&gt;     &lt;xsl:otherwise&gt;unknown&lt;/xsl:otherwise&gt;   &lt;/xsl:choose&gt; &lt;/xsl:variable&gt;</pre>
<b>Usage Examples</b>	See <i>Example: Configuring Dual Routing Engines</i> , <i>Example: Preventing Import of the Full Routing Table</i> , and <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">SLAX Statements Overview on page 100</a></li><li>• <a href="#">else on page 203</a></li><li>• <a href="#">else if on page 204</a></li><li>• <a href="#">for-each on page 208</a></li></ul>

---

## import

---

<b>Syntax</b>	<code>import <i>href</i>;</code>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	<p>Import rules from an external file or style sheet, which provide access to all the declarations and templates within the imported item. Any <b>import</b> statements must be the first elements in the script or style sheet. The path can be any URI. The path <code>../import/junos.xsl</code> is standard for all commit scripts, op scripts, and event scripts.</p> <p>Imported rules are overwritten by any subsequent matching rules within the importing script. If more than one file or style sheet is imported, the items imported last override each previous import where the rules match.</p>
<b>Attributes</b>	<code>href</code> —Specifies the location of the imported file or style sheet.
<b>SLAX Example</b>	<p>In the example, the main script imports the file <code>route-rules.slax</code>, which contains a template rule for <code>&lt;route&gt;</code> elements.</p> <pre>version 1.1; import "route-rules.slax";</pre>

```

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

```

**Related Documentation**

- [Junos Script Automation: Named Templates in the jcs Namespace Overview on page 40](#)
- [Junos Script Automation: Global Parameters and Variables in the junos.xml File on page 51](#)
- *Required Boilerplate for Commit Scripts*
- *Required Boilerplate for Event Scripts*
- *Required Boilerplate for Op Scripts*
- [apply-imports on page 195](#)

## key

**Syntax**

```

key name {
  match pattern;
  value expression;
}

```

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

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

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

For example, if you define the key:

```

key func {
  match prototype;
  value @name;
}

```

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

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

**Attributes**    *key name*—Key identifier, which uniquely identifies the key within the script and is passed as the first argument to the **key()** function.

*match pattern*—XPath expression that selects the set of nodes to index.

*value expression*—XPath expression that defines the value of the key.

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

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

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

```
version 1.1;  
  
ns junos = "http://xml.juniper.net/junos/*/junos";  
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";  
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";  
  
key protocol {  
  match route-table/rt;  
  value rt-entry/protocol-name;  
}  
key next-hop {  
  match route-table/rt;  
  value rt-entry/nh/via;  
}  
  
match / {  
  <op-script-results> {  
    var $results = jcs:invoke("get-route-information");  
  
    for-each( $results ) {  
      /* Display all static routes */  
      <output> "Static routes: ";
```

```

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

```

- Related Documentation**
- [XPath Overview on page 65](#)
  - [XPath Expressions Overview for SLAX on page 87](#)

## match

<b>Syntax</b>	<pre> match <i>expression</i> {   <i>statements</i>; } </pre>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	Declare a template that contains rules to apply when a specified node is matched. The <b>match</b> statement associates the template with an XML element. The <b>match</b> statement can also be used to define a template for a whole branch of the XML document. For example, <b>match /</b> matches the root element of the document.
<b>Attributes</b>	<i>expression</i> —XPath expression specifying the nodes to which to apply the template.
<b>SLAX Example</b>	<pre> match host-name {   &lt;hello&gt; .; } </pre>
<b>XSLT Equivalent</b>	<pre> &lt;xsl:template match="host-name"&gt;   &lt;hello&gt;     &lt;xsl:value-of select="."/&gt;   &lt;/hello&gt; &lt;/xsl:template&gt; </pre>
<b>Usage Examples</b>	<i>Example: Adding a Final then accept Term to a Firewall, Example: Configuring Dual Routing Engines, and Example: Preventing Import of the Full Routing Table.</i>
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">apply-templates on page 195</a></li> <li>• <a href="#">call on page 199</a></li> <li>• <a href="#">mode on page 216</a></li> <li>• <a href="#">priority on page 226</a></li> <li>• <a href="#">template on page 232</a></li> </ul>

- [with on page 239](#)

---

## message

---

<b>Syntax</b>	<pre>message expression;  message {     /* body */ }</pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	<p>Generate an error message that is immediately displayed to the user, typically on the standard error file descriptor. This is different from most script output, which is displayed only after the script generates the final result tree.</p> <p>Junos OS op scripts, event scripts, and commit scripts prepend "error:" to the displayed message when generating text output. When generating XML output, the scripts place the output inside a <b>&lt;message&gt;</b> element, which is enclosed in an <b>&lt;xmn:error&gt;</b> element.</p> <p>If the <b>message</b> statement is used in a commit script, the script will generate two errors and terminate the commit process. If the <b>message</b> statement is used in an event script, the script writes the message to the output file, if one is configured.</p>
<b>Attributes</b>	<i>message expression</i> —XPath expression or string emitted as output.
<b>SLAX Example</b>	<pre>if (not(valid)) {     message "The " _ name() _ " node is not valid"; }</pre>

---

## mode

---

<b>Syntax</b>	<pre>mode qualified-name;</pre>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	<p>Indicate the mode in which a template needs to be applied for the template to be used. If templates are applied in the specified mode, the <b>match</b> statement is used to determine whether the template can be used with the particular node. If more than one template matches a node in the specified mode, the priority statement determines which template is used. The highest priority wins. If no priority is specified explicitly, the priority of a template is determined by the <b>match</b> statement.</p> <p>This statement is comparable to the <b>mode</b> attribute of the <b>&lt;xsl:template&gt;</b> element. You can include this statement inside a SLAX <b>match</b> or <b>apply-templates</b> statement.</p>
<b>SLAX Example</b>	<pre>match * {     mode "one";     &lt;one&gt; .; }</pre>

```

match * {
  mode "two";
  <two> string-length(.);
}

match / {
  apply-templates version {
    mode "one";
  }
  apply-templates version {
    mode "two";
  }
}

```

**XSLT Equivalent**

```

<xsl:template match="*" mode="one">
  <one>
    <xsl:value-of select="."/>
  </one>
</xsl:template>

<xsl:template match="*" mode="two">
  <two>
    <xsl:value-of select="string-length(.)"/>
  </two>
</xsl:template>

<xsl:template match="/">
  <xsl:apply-templates select="version" mode="one"/>
  <xsl:apply-templates select="version" mode="two"/>
</xsl:template>

```

**Usage Examples**    See *Example: Adding a Final then accept Term to a Firewall*.

**Related Documentation**

- [apply-templates on page 195](#)
- [call on page 199](#)
- [match on page 215](#)
- [priority on page 226](#)
- [template on page 232](#)
- [with on page 239](#)
- [xsl:template on page 256](#)

**mvar**

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

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

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

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



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

---

**Attributes** *name*—Mutable variable identifier. After declaration, you can reference the variable within expressions by using the identifier prefixed with the dollar sign (\$) character.

*initial-value*—Initial value assigned to the mutable variable.

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

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

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

**Related Documentation**

- [SLAX Variables Overview on page 97](#)
- [append on page 194](#)
- [set on page 230](#)
- [var on page 237](#)

---

## number

**Syntax**

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

number {
  count nodes;
  format numbering-style;
```



```

    from nodes;
    grouping-separator character;
    grouping-size number;
    level "single" | "multiple" | "any";
}

```

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

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

**Attributes** *number expression*—XPath expression providing the number to format.

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

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

The **format** option can include the following:

- **start string**—Any non-alphanumeric characters that precede the first number token in the format string. The start string is prepended to the formatted number string.
- **number token**—One or more number tokens that indicate what numbering format to use for the included numbers. The formatted number string only includes more than one number if the **level** option is set to "multiple". [Table 25 on page 219](#) lists format values and corresponding styles. The default value is "1", which uses a decimal format style. When using decimal format, you can specify the minimum length of the formatted number string by preceding the "1" with one or more zeros.
- **token separator**—Non-alphanumeric characters that separate number tokens in the format string. These characters are included in the formatted number string between the computed numbers.
- **end string**—Any non-alphanumeric characters that follow the last number token in the format string. The end string is appended to the formatted number string.

**Table 25: Numbering Styles for SLAX Statement *number*, *format* Option**

Format Value	Style	Example
1	Decimal format	1 2 3 ...10 11 ...
01	Decimal format with a minimum output string length of 2	01 02 03 ... 10 11 ...

**Table 25: Numbering Styles for SLAX Statement `number, format` Option (*continued*)**

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

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

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

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

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

- **single**—Perform only one count. The current node, if it matches the count expression, or the nearest ancestor that matches the count expression, is counted. The position of the node in document order, relative to its siblings that also match the count parameter, is used as the number to be formatted.
- **multiple**—Separately count all nodes that match the count expression and are either the current node or an ancestor of the current node. The position of each node in document order, relative to its siblings that also match the count parameter, is used as one of the numbers to be formatted.
- **any**—Perform only one count. The current node, if it matches the count pattern, or its nearest ancestor or preceding node that matches the count pattern, is counted. The position of the node in document order, relative to all other matching nodes that are ancestors or precede the node, is used as the number to be formatted.



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

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

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

I II III IV V

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

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

1,234,567,890

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

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

```
var $data := {
  <interfaces> {
    <interface> {
      <name> "ge-0/0/0";
      <unit> {
        <name> "0";
      }
      <unit> {
        <name> "1";
      }
    }
  }
  <interface> {
    <name> "ge-0/1/0";
    <unit> {
```

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

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

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

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

- Related Documentation
- [decimal-format on page 201](#)
  - [output-method on page 222](#)

---

## output-method

Syntax	<pre>output-method <i>output-format</i> {   cdata-section-elements <i>name-list</i>;   doctype-public <i>string</i>;   doctype-system <i>string</i>;   encoding <i>string</i>;</pre>
--------	--

```

    indent "yes" | "no";
    media-type string;
    omit-xml-declaration "yes" | "no";
    standalone "yes" | "no";
    version string;
}

```

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

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

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

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

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

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

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

*indent "yes" | "no"*—Specify whether to indent the result tree output according to the hierarchical structure. Acceptable values are “yes” and “no”. The default is no indentation.

*media-type string*—Define the MIME content type of the output. The default is “text/xml”.

*omit-xml-declaration "yes" | "no"*—Specify whether to include or omit the XML declaration (**<?xml ?>**) in the output. The default is “no”.

*standalone "yes" | "no"*—Explicitly add the pseudo-attribute **standalone** with the given string value to the XML declaration (**<?xml ?>**) in the output. Acceptable values are “yes” and “no”. The **standalone** attribute is only relevant if the document uses a

DTD. If the **standalone** option is not included in the **output-method** statement, there is no explicit declaration in the result tree, which is identical to **standalone="no"**.

*version string*—For HTML and XML formats, set the W3C version for the output format.

The pseudo-attribute **version** is included in the XML declaration (`<?xml ?>`) with the given version number.

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

```
version 1.1;

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

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

The script produces the following output:

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

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

```
version 1.1;

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

```

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

```

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

```

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

```

## param

<b>Syntax</b>	<code>param \$name=value;</code>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	<p>Declare a parameter for a template or for the style sheet as a whole. Template parameters declared with the <b>param</b> statement must be placed inside the template code block. A global parameter, the scope of which is the entire style sheet, must be declared at the top level of the style sheet. You can include an initial value by following the parameter name with an equal sign (=) and a value expression. A parameter whose value is set by Junos OS at script initialization must be defined as a global parameter.</p> <p>In SLAX, parameter and variable names are declared and accessed using the dollar sign (\$). This is unlike the <b>name</b> attribute of <code>&lt;xsl:variable&gt;</code> and <code>&lt;xsl:parameter&gt;</code> elements, which do not include the dollar sign in the declaration.</p>
<b>Attributes</b>	<p><i>name</i>—Defines the name of the parameter.</p> <p><i>value</i>—Defines the default value for the parameter, which is used if the person or client application that executes the script does not explicitly provide a value.</p>
<b>SLAX Example</b>	<pre> param \$vrf; param \$dot = .; </pre>
<b>XSLT Equivalent</b>	<pre> &lt;xsl:param name="vrf"/&gt; &lt;xsl:param name="dot" select="."/&gt; </pre>
<b>Usage Examples</b>	<ul style="list-style-type: none"> <li>• <i>Example: Imposing a Minimum MTU Setting</i></li> <li>• <i>Example: Limiting the Number of ATM Virtual Circuits</i></li> <li>• <i>Example: Limiting the Number of E1 Interfaces</i></li> <li>• <i>Example: Preventing Import of the Full Routing Table</i></li> <li>• <i>Example: Requiring and Restricting Configuration Statements</i></li> </ul>
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">SLAX Parameters Overview on page 93</a></li> </ul>

- [SLAX Templates Overview on page 88](#)
- [template on page 232](#)
- [var on page 237](#)
- [with on page 239](#)

---

## preserve-space

---

<b>Syntax</b>	<code>preserve-space <i>element-list</i>;</code>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	<p>Preserve whitespace-only child text nodes for the source tree element nodes listed, but not for the child text nodes of the element node children. To preserve whitespace-only child text nodes of the element node children, specify the child nodes as separate entries in the preserve-space element list. Specifying an asterisk preserves whitespace-only child elements for all elements, which is the default behavior. A text node is considered whitespace-only if it includes only spaces, tabs, newlines, and carriage returns.</p> <p>The <b>preserve-space</b> statement is only needed if the <b>strip-space</b> statement has been used with an asterisk, indicating that whitespace-only child text nodes should be removed from all element nodes. In this case, use the <b>preserve-space</b> statement to indicate specific element nodes that should not have their whitespace-only child text nodes stripped.</p> <p>This statement must be defined as a top-level statement in the script.</p>
<b>Attributes</b>	<i>element-list</i> —Space-separated list of element names for which to preserve whitespace-only child text nodes.
<b>SLAX Example</b>	<p>The following example removes all whitespace-only text nodes from the source tree except for child elements of <b>&lt;user-context&gt;</b>:</p> <pre>version 1.1;  preserve-space user-context; strip-space *;  match / { ... }</pre>
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">strip-space on page 232</a></li></ul>

---

## priority

---

<b>Syntax</b>	<code>priority <i>number</i>;</code>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.



**Description** If more than one template matches a node in the specified mode, this statement determines which template is used. The highest priority wins. If no priority is specified explicitly, the priority of a template is determined by the **match** statement.

This statement is comparable to the **priority** attribute of the `<xsl:template>` element. You can include this statement inside a SLAX **match** statement.

**SLAX Example**

```
match * {
  priority 10;
  <output> .;
}
```

**XSLT Equivalent**

```
<xsl:template match="*" priority="10">
  <output>
    <xsl:value-of select="."/>
  </output>
</xsl:template>
```

**Usage Examples** None of the examples in this manual use this statement.

- Related Documentation**
- [apply-templates on page 195](#)
  - [call on page 199](#)
  - [match on page 215](#)
  - [mode on page 216](#)
  - [template on page 232](#)
  - [with on page 239](#)
  - [xsl:template on page 256](#)

## processing-instruction

**Syntax**

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

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

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

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

**Attributes** *instruction-name*—Identifier for the processing instruction, which can be a string, a variable, or an XPath expression.

*instruction-value*—Instruction content, which consists of name-value pairs.

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

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

The corresponding output in the result tree is:

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

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

```
version 1.1;

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

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

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

## result

<b>Syntax</b>	<pre> result expression;  result {   /* body */ } </pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	Define the return value for a function. The value can be a simple scalar value, an XML element or XPath expression, or a set of instructions that emit the value to be returned.
<b>Attributes</b>	<i>result expression</i> —XPath expression defining the return value of the function.
<b>SLAX Example</b>	<p>The following example defines three extension functions, <b>my:size()</b>, <b>my:box-parts()</b>, and <b>my:ark()</b>. The <b>my:ark()</b> function returns a node set containing an <b>&lt;ark&gt;</b> element that encloses the node set returned by the <b>my:box-parts()</b> function. The <b>my:box-parts()</b> function returns a node set containing a <b>&lt;box&gt;</b> element enclosing three <b>&lt;part&gt;</b> child elements. The content of each <b>&lt;part&gt;</b> element is the value returned by the <b>my:size()</b> function. The return value of the <b>my:size()</b> function is the product of the three parameters <b>width</b>, <b>height</b>, and <b>scale</b>.</p> <pre> version 1.1; ns my exclude = "http://www.example.com/myfunctions";  function my:size (\$x, \$y, \$scale = 1) {   result \$x * \$y * \$scale; } function my:box-parts (\$width, \$height, \$depth, \$scale = 1) {   result &lt;box&gt; {     &lt;part count=2&gt; my:size(\$width, \$depth);     &lt;part count=2&gt; my:size(\$width, \$height);     &lt;part count=2&gt; my:size(\$depth, \$height);   } } function my:ark () {   result {     &lt;ark&gt; {       copy-of my:box-parts(2.5, 1.5, 1.5);     }   } }  match / {   var \$res = my:ark();   copy-of \$res; } </pre>
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">SLAX Functions Overview on page 91</a></li> <li>• <a href="#">copy-of on page 201</a></li> </ul>

- [function on page 210](#)

---

## set

---

<b>Syntax</b>	<code>set \$name = value;</code>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	Assign a value to a mutable variable. The variable must be defined using the <b>mvar</b> statement.
<b>Attributes</b>	<i>name</i> —Name of the mutable variable.  <i>value</i> —Value to assign to the mutable variable.
<b>SLAX Example</b>	<p>The following example creates the mutable variable, <b>block</b>. The <b>set</b> statement assigns an initial value to the <b>block</b> variable. In the <b>for</b> loop, the code iterates over each item in the specified list and appends an <b>&lt;item&gt;</b> element with two child elements, <b>&lt;name&gt;</b> and <b>&lt;size&gt;</b>, to the node set stored in the <b>block</b> variable.</p> <pre>mvar \$block; set \$block = &lt;block&gt; "item list";  for \$item (list) {   append \$block += &lt;item&gt; {     &lt;name&gt; \$item/name;     &lt;size&gt; \$item/size;   } }</pre>
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">SLAX Variables Overview on page 97</a></li><li>• <a href="#">append on page 194</a></li><li>• <a href="#">mvar on page 217</a></li><li>• <a href="#">var on page 237</a></li></ul>

---

## sort

---

<b>Syntax</b>	<pre>sort expression {;   case-order "upper-first"   "lower-first";   data-type "text"   "number"   type-name;   order "ascending"   "descending"; }</pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	Control the order in which the <b>for-each</b> and <b>apply-templates</b> statements iterate through the current node list. By default, the <b>for-each</b> and <b>apply-templates</b> statements consider nodes in document order, but the <b>sort</b> statement defines the order prior to iterating

through the node list. Insert the **sort** statement immediately after the **for-each** or **apply-templates** statement. The **sort** statement is only processed when the loop is first initiated.

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

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

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

*case-order*—Specify whether to sort lowercase first or uppercase first. Acceptable values are "lower-first" or "upper-first". The default is "upper-first".

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

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

*order*—Specify whether to sort in ascending or descending order. Acceptable values are "descending" or "ascending". The default is "ascending".

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

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

match / {
  <op-script-results> {

    var $results = jcs:invoke("get-interface-information");
```

```
    for-each ($results/physical-interface/name) {
      sort . {
        data-type "text";
        order "ascending";
      }
      <interface-name> .;
    }
  }
}
```

- Related Documentation
- [apply-templates on page 195](#)
  - [for-each on page 208](#)

---

## strip-space

Syntax	<code>strip-space <i>element-list</i>;</code>
Release Information	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
Description	<p>Remove whitespace-only child text nodes from the source tree element nodes listed, but not from the child text nodes of the element node children. To perform whitespace stripping on the child text nodes of the element node children, specify the child nodes as separate entries in the strip-space element list. Specifying an asterisk removes whitespace-only child elements from all elements. A text node is considered whitespace-only if it includes only spaces, tabs, newlines, and carriage returns.</p> <p>This statement must be defined as a top-level statement in the script. The default is to preserve all whitespace-only elements.</p>
Attributes	<i>element-list</i> —List of element names separated by spaces.
SLAX Example	<p>The following example removes all whitespace-only text nodes from the source tree except for child elements of <code>&lt;user-context&gt;</code>:</p> <pre>version 1.1;  preserve-space user-context; strip-space *;  match / {   ... }</pre>
Related Documentation	• <a href="#">preserve-space on page 226</a>

---

## template

Syntax	<code>template <i>qualified-name</i> (<i>parameter-name</i> = <i>value</i>) {</code> <i>/* code */</i> }
--------	--

<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	Declare a named template. You can include a comma-separated list of parameter declarations, with the parameter name and an optional equal sign (=) and value expression. You can declare additional parameters inside the code block using the <b>param</b> statement. You can invoke the template using the <b>call</b> statement.
<b>SLAX Example</b>	<pre> match configuration {   var \$name-servers = name-servers/name;   call temp();   call temp(\$name-servers, \$size = count(\$name-servers));   call temp() {     with \$name-servers;     with \$size = count(\$name-servers);   }    template temp(\$name-servers, \$size = 0) {     &lt;output&gt; "template called with size " _ \$size;   } } </pre>
<b>XSLT Equivalent</b>	<pre> &lt;xsl:template match="configuration"&gt;   &lt;xsl:variable name="name-servers" select="name-servers/name"/&gt;   &lt;xsl:call-template name="temp"/&gt;   &lt;xsl:call-template name="temp"&gt;     &lt;xsl:with-param name="name-servers" select="\$name-servers"/&gt;     &lt;xsl:with-param name="size" select="count(\$name-servers)"/&gt;   &lt;/xsl:call-template&gt;   &lt;xsl:call-template name="temp"&gt;     &lt;xsl:with-param name="name-servers" select="\$name-servers"/&gt;     &lt;xsl:with-param name="size" select="count(\$name-servers)"/&gt;   &lt;/xsl:call-template&gt; &lt;/xsl:template&gt;  &lt;xsl:template name="temp"&gt;   &lt;xsl:param name="name-servers"/&gt;   &lt;xsl:param name="size" select="0"/&gt;   &lt;output&gt;     &lt;xsl:value-of select="concat('template called with size ', \$size)"/&gt;   &lt;/output&gt; &lt;/xsl:template&gt; </pre>
<b>Usage Examples</b>	See <i>Example: Adding a Final then accept Term to a Firewall</i> and <i>Example: Adding T1 Interfaces to a RIP Group</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">SLAX Parameters Overview on page 93</a></li> <li>• <a href="#">SLAX Templates Overview on page 88</a></li> <li>• <a href="#">apply-templates on page 195</a></li> <li>• <a href="#">call on page 199</a></li> <li>• <a href="#">match on page 215</a></li> </ul>

- [mode on page 216](#)
- [priority on page 226](#)
- [with on page 239](#)

---

## terminate

---

<b>Syntax</b>	<pre>terminate <i>expression</i>;  terminate {     /* body */ }</pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	<p>Generate an error message that is immediately displayed to the user, and exit the script.</p> <p>Junos OS op scripts, event scripts, and commit scripts prepend "error:" to the displayed message when generating text output. When generating XML output, the scripts place the output inside a <b>&lt;message&gt;</b> element, which is enclosed in an <b>&lt;xmn:error&gt;</b> element.</p> <p>If the <b>terminate</b> statement is used in a commit script, the script will generate two errors and terminate the script and the commit process. If the <b>terminate</b> statement is used in an event script, the script writes the message to the output file, if one is configured, and terminates the script.</p>
<b>Attributes</b>	<i>expression</i> —XPath expression or string emitted as output.
<b>SLAX Example</b>	<pre>if (not(valid)) {     terminate "The "_name()_" node is not valid. Exiting script." }</pre>
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">message on page 216</a></li></ul>

---

## trace

---

<b>Syntax</b>	<pre>trace <i>expression</i>;  trace {     /* body */ }</pre>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	Write a message to the trace file, if tracing is enabled. If tracing is not enabled, trace output is not generated. The <b>trace</b> message can be an XPath expression or string, or it can be generated by the contents of a <b>trace</b> statement block.



Enabling of tracing is typically a feature of the environment in which a SLAX script is called. When executing a script using the **slaxproc** command, include the **--trace** or **-t** option to enable tracing. For more information about slaxproc, see [“Understanding the SLAX Processor \(slaxproc\)” on page 111](#).

**Attributes** *trace expression*—XPath expression or string written to the trace file.

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

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

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

trace {
  if ($my-trace-flag) {
    expr "max " _ $max _ "; min " _ $min;
    copy-of options;
  }
}
```

**Related Documentation**

- [message on page 216](#)
- [terminate on page 234](#)
- [Understanding the SLAX Processor \(slaxproc\) on page 111](#)

## uexpr

<b>Syntax</b>	<code>uexpr <i>expression</i>;</code>
<b>Release Information</b>	Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.
<b>Description</b>	Generate the string value of an XPath expression and add it to the result tree, but do not escape special characters. The <b>uexpr</b> statement behaves identically to the <b>expr</b> statement, except that the contents are not escaped. By default, characters such as "<", ">", and "&" are escaped into proper XML as "&lt;", "&gt;", and "&amp;", respectively, but <b>uexpr</b> does not execute this escaping mechanism.
<b>Attributes</b>	<i>expression</i> —XPath expression to add to the result tree.

**SLAX Example** The following statement outputs the string to the result tree exactly as it appears in the statement. If **expr** is used in place of **uexpr**, the script would output the string "&lt;&amp;&gt;".

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

- Related Documentation**
- [expr on page 205](#)
  - [message on page 216](#)
  - [terminate on page 234](#)
  - [trace on page 234](#)

---

## use-attribute-sets

---

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

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

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

**Attributes** *attribute-set-name*—Name of the attribute set, which is defined using an **attribute-set** statement.

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

```
version 1.1;

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

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

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

```
    }
  }
```

- Related Documentation**
- [SLAX Elements and Element Attributes Overview on page 85](#)
  - [attribute on page 196](#)
  - [attribute-set on page 197](#)
  - [element on page 203](#)

## var

<b>Syntax</b>	<code>var \$name=value;</code>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	Declare a local or global variable. A variable is global if it is defined outside of any template. Otherwise, it is local. The value of a global variable is accessible anywhere in the style sheet. The scope of a local variable is limited to the template or code block in which it is defined. Variables declared in this manner are immutable. You initialize a variable by following the variable name with an equal sign (=) and an expression.
<b>Attributes</b>	<p><i>name</i>—Specifies the name of the variable. After declaration, the variable can be referred to within expressions using this name, including the \$ character.</p> <p><i>value</i>—Defines the default value for the variable, which is used if the person or client application that executes the script does not explicitly provide a value.</p>
<b>SLAX Example</b>	<pre>var \$vrf; var \$location = \$dot/@location; var \$message = "We are in "_\$location_" now.";</pre>
<b>XSLT Equivalent</b>	<pre>&lt;xsl:variable name="vrf"/&gt; &lt;xsl:variable name="location" select="\$dot/location"/&gt; &lt;xsl:variable name="message" select="concat('We are in ', \$location, now.)"/&gt;</pre>
<b>Usage Examples</b>	See <i>Example: Limiting the Number of E1 Interfaces</i> , <i>Example: Limiting the Number of ATM Virtual Circuits</i> , <i>Example: Configuring Administrative Groups for LSPs</i> , and <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">SLAX Variables Overview on page 97</a></li> <li>• <a href="#">mvar on page 217</a></li> <li>• <a href="#">param on page 225</a></li> </ul>

## version

<b>Syntax</b>	<code>version 1.0;</code>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.

<b>Description</b>	<p>Specify the version of SLAX that is being used. All SLAX style sheets must begin with a <b>version</b> statement.</p> <p>Version 1.0 uses XML version 1.0 and XSLT version 1.1.</p> <p>In addition, the <b>xsl</b> namespace is implicitly defined as follows:</p> <pre>xmlns:xsl="http://www.w3.org/1999/XSL/Transform"</pre>
<b>Attributes</b>	<p><i>version-number</i>—Specifies the version of SLAX. Junos OS supports SLAX version 1.0.</p>
<b>SLAX Example</b>	<pre>version 1.0;</pre>
<b>XSLT Equivalent</b>	<pre>&lt;xsl:stylesheet version="1.0"&gt;</pre>
<b>Usage Examples</b>	<ul style="list-style-type: none"><li>• <i>Example: Adding a Final then accept Term to a Firewall</i></li><li>• <i>Example: Changing the Configuration Using an Op Script</i></li><li>• <i>Example: Assigning a Classifier</i></li><li>• <i>Example: Imposing a Minimum MTU Setting</i></li><li>• <i>Example: Restarting an FPC Using an Op Script</i></li></ul>
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <i>Required Boilerplate for Commit Scripts</i></li><li>• <i>Required Boilerplate for Event Scripts</i></li><li>• <i>Required Boilerplate for Op Scripts</i></li><li>• <a href="#">SLAX Syntax Rules Overview on page 83</a></li></ul>

---

## while

<b>Syntax</b>	<pre>while (expression) {     /* body */ }</pre>
<b>Release Information</b>	<p>Statement introduced in version 1.1 of the SLAX language, which is supported in Junos OS Release 12.2 and later releases.</p>
<b>Description</b>	<p>Repeatedly execute a block of statements until the specified condition evaluates to <b>false</b>. The condition is an XPath expression that is converted to a boolean type. If the expression evaluates to <b>true</b>, the contents of the while loop are executed. The loop continues to execute until the expression evaluates to <b>false</b>. During execution, the context is not changed. In the expression, you should use a mutable variable, which is declared using the <b>mvar</b> statement, to avoid creating an infinite loop.</p>
<b>Attributes</b>	<p><i>expression</i>—XPath expression, which is cast to boolean type and used as the condition for the while loop. The code block contents are executed as long as the condition evaluates to <b>true</b>.</p>

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

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

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

**Related Documentation**

- [SLAX Variables Overview on page 97](#)
- [XPath Overview on page 65](#)
- [mvar on page 217](#)

## with

<b>Syntax</b>	<code>with \$name = value;</code>
<b>Release Information</b>	Statement introduced in version 1.0 of the SLAX language.
<b>Description</b>	<p>Specify a parameter to pass into a template. You can use this statement when you apply templates with the <b>apply-templates</b> statement or invoke templates with the <b>call</b> statement.</p> <p>Optionally, you can specify a value for the parameter by including an equal sign (=) and a value expression. If no value is specified, the current value of the parameter is passed to the template.</p>
<b>Attributes</b>	<p><i>name</i>—Name of the variable or parameter for which the value is being passed.</p> <p><i>value</i>—Value of the parameter being passed to the template.</p>
<b>SLAX Example</b>	<pre>match configuration {   var \$domain = domain-name;   apply-templates system/host-name {     with \$message = "Invalid host-name";     with \$domain;   } }  match host-name {   param \$message = "Error";   param \$domain;   &lt;hello&gt; \$message _ "::" _ . _ " (" _ \$domain _ ")"; }</pre>
<b>XSLT Equivalent</b>	<pre>&lt;xsl:template match="configuration"&gt;   &lt;xsl:apply-templates select="system/host-name"&gt;     &lt;xsl:with-param name="message" select="'Invalid host-name'"/&gt;   &lt;/xsl:apply-templates&gt; &lt;/xsl:template&gt;</pre>

```
<xsl:with-param name="domain" select="$domain"/>
</xsl:apply-templates>
</xsl:template>

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

**Usage Examples**    See *Example: Configuring Dual Routing Engines*, *Example: Preventing Import of the Full Routing Table*, and *Example: Automatically Configuring Logical Interfaces and IP Addresses*.

- Related Documentation**
- [SLAX Parameters Overview on page 93](#)
  - [SLAX Templates Overview on page 88](#)
  - [apply-templates on page 195](#)
  - [call on page 199](#)
  - [match on page 215](#)
  - [mode on page 216](#)
  - [priority on page 226](#)
  - [template on page 232](#)

## CHAPTER 16

# Standard XPath and XSLT Functions Used in Automation Scripts

- [concat\(\)](#) on page 241
- [contains\(\)](#) on page 242
- [count\(\)](#) on page 242
- [last\(\)](#) on page 242
- [name\(\)](#) on page 243
- [not\(\)](#) on page 243
- [position\(\)](#) on page 243
- [starts-with\(\)](#) on page 244
- [string-length\(\)](#) on page 244
- [substring-after\(\)](#) on page 245
- [substring-before\(\)](#) on page 245

### concat()

---

<b>Syntax</b>	<code>string concat(string, string+)</code>
<b>Description</b>	Return the concatenation of the arguments.
<b>Usage Examples</b>	See <i>Example: Limiting the Number of E1 Interfaces</i> , <i>Example: Controlling IS-IS and MPLS Interfaces</i> , <i>Example: Adding T1 Interfaces to a RIP Group</i> , <i>Example: Configuring Administrative Groups for LSPs</i> , and <i>Example: Configuring Dual Routing Engines</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">contains()</a> on page 242</li><li>• <a href="#">starts-with()</a> on page 244</li><li>• <a href="#">string-length()</a> on page 244</li><li>• <a href="#">substring-after()</a> on page 245</li><li>• <a href="#">substring-before()</a> on page 245</li></ul>

## contains()

---

<b>Syntax</b>	<i>boolean</i> contains( <i>string</i> , <i>string</i> )
<b>Description</b>	Return TRUE if the first string argument contains the second string argument, otherwise return FALSE.
<b>Usage Examples</b>	See <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">concat()</a> on page 241</li><li>• <a href="#">starts-with()</a> on page 244</li><li>• <a href="#">string-length()</a> on page 244</li><li>• <a href="#">substring-after()</a> on page 245</li><li>• <a href="#">substring-before()</a> on page 245</li></ul>

## count()

---

<b>Syntax</b>	<i>number</i> count( <i>node-set</i> )
<b>Description</b>	Return the number of nodes in the argument node-set.
<b>Usage Examples</b>	See <i>Example: Limiting the Number of E1 Interfaces</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">last()</a> on page 242</li><li>• <a href="#">name()</a> on page 243</li><li>• <a href="#">not()</a> on page 243</li><li>• <a href="#">position()</a> on page 243</li></ul>

## last()

---

<b>Syntax</b>	<i>number</i> last()
<b>Description</b>	Return the index of the last node in the list that is currently being evaluated, which is equal to the number of items in the processed node list.
<b>Usage Examples</b>	See <i>Example: Limiting the Number of E1 Interfaces</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">count()</a> on page 242</li><li>• <a href="#">name()</a> on page 243</li><li>• <a href="#">not()</a> on page 243</li><li>• <a href="#">position()</a> on page 243</li></ul>



## name()

<b>Syntax</b>	<i>string</i> name(<node-set>)
<b>Description</b>	Return the full name of the first node in the node set, including the prefix for its namespace declared in the source document. If no argument is passed, the function returns the full name of the context node.
<b>Usage Examples</b>	See <a href="#">jcs:emit-change Template</a> .
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">count()</a> on page 242</li> <li>• <a href="#">last()</a> on page 242</li> <li>• <a href="#">not()</a> on page 243</li> <li>• <a href="#">position()</a> on page 243</li> </ul>

## not()

<b>Syntax</b>	<i>boolean</i> not( <i>boolean</i> )
<b>Description</b>	Return TRUE if the argument is FALSE, and FALSE if the argument is TRUE.
<b>Usage Examples</b>	See <i>Example: Requiring and Restricting Configuration Statements</i> , <i>Example: Controlling IS-IS and MPLS Interfaces</i> , <i>Example: Configuring a Default Encapsulation Type</i> , <i>Example: Controlling LDP Configuration</i> , <i>Example: Adding a Final then accept Term to a Firewall</i> , <i>Example: Configuring Administrative Groups for LSPs</i> , <i>Example: Configuring Dual Routing Engines</i> , and <i>Example: Preventing Import of the Full Routing Table</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">count()</a> on page 242</li> <li>• <a href="#">last()</a> on page 242</li> <li>• <a href="#">name()</a> on page 243</li> <li>• <a href="#">position()</a> on page 243</li> </ul>

## position()

<b>Syntax</b>	<i>number</i> position()
<b>Description</b>	Return the position of the context node among the list of nodes that are currently being evaluated.
<b>Usage Examples</b>	See <i>Example: Adding a Final then accept Term to a Firewall</i> and <i>Example: Prepending a Global Policy</i> .

- Related Documentation**
- [count\(\) on page 242](#)
  - [last\(\) on page 242](#)
  - [name\(\) on page 243](#)
  - [not\(\) on page 243](#)

---

## starts-with()

---

- |                              |  |
|------------------------------|--|
| <b>Syntax</b>                | <i>boolean starts-with(string, string)</i>   |
| <b>Description</b>           | Return TRUE if the first string argument starts with the second string argument, otherwise return FALSE.   |
| <b>Usage Examples</b>        | See <i>Example: Imposing a Minimum MTU Setting</i> , <i>Example: Limiting the Number of E1 Interfaces</i> , <i>Example: Limiting the Number of ATM Virtual Circuits</i> , <i>Example: Adding T1 Interfaces to a RIP Group</i> , <i>Example: Configuring a Default Encapsulation Type</i> , and <i>Example: Configuring Dual Routing Engines</i> .  |
| <b>Related Documentation</b> | <ul style="list-style-type: none"><li>• <a href="#">concat() on page 241</a></li><li>• <a href="#">contains() on page 242</a></li><li>• <a href="#">string-length() on page 244</a></li><li>• <a href="#">substring-after() on page 245</a></li><li>• <a href="#">string-length() on page 244</a></li><li>• <a href="#">substring-after() on page 245</a></li><li>• <a href="#">substring-before() on page 245</a></li></ul> |

---

## string-length()

---

- |                              |  |
|------------------------------|--|
| <b>Syntax</b>                | <i>number string-length(&lt;string&gt;)</i>  |
| <b>Description</b>           | Return the number of characters in the string. If the argument is omitted, it returns the string value of the context node.  |
| <b>Usage Examples</b>        | See <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .  |
| <b>Related Documentation</b> | <ul style="list-style-type: none"><li>• <a href="#">concat() on page 241</a></li><li>• <a href="#">contains() on page 242</a></li><li>• <a href="#">starts-with() on page 244</a></li><li>• <a href="#">substring-after() on page 245</a></li><li>• <a href="#">substring-before() on page 245</a></li></ul> |

## substring-after()

---

<b>Syntax</b>	<code>string substring-after(string, string)</code>
<b>Description</b>	Return the portion of the first string argument that follows the occurrence of the second argument substring within the first. If the second string is not contained in the first string, or if the second string is empty, the function returns an empty string.
<b>Usage Examples</b>	See <i>Example: Limiting the Number of EI Interfaces</i> and <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">concat() on page 241</a></li><li>• <a href="#">contains() on page 242</a></li><li>• <a href="#">starts-with() on page 244</a></li><li>• <a href="#">string-length() on page 244</a></li><li>• <a href="#">substring-before() on page 245</a></li></ul>

## substring-before()

---

<b>Syntax</b>	<code>string substring-before(string, string)</code>
<b>Description</b>	Return the portion of the first string argument that precedes the occurrence of the second argument substring within the first. If the second string is not contained in the first string, or if the second string is empty, the function returns an empty string.
<b>Usage Examples</b>	See <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">concat() on page 241</a></li><li>• <a href="#">contains() on page 242</a></li><li>• <a href="#">starts-with() on page 244</a></li><li>• <a href="#">string-length() on page 244</a></li><li>• <a href="#">substring-after() on page 245</a></li></ul>



## CHAPTER 17

# Standard XSLT Elements and Attributes Used in Automation Scripts

- [xsl:apply-templates on page 248](#)
- [xsl:call-template on page 248](#)
- [xsl:choose on page 249](#)
- [xsl:comment on page 250](#)
- [xsl:copy-of on page 250](#)
- [xsl:element on page 251](#)
- [xsl:for-each on page 251](#)
- [xsl:if on page 252](#)
- [xsl:import on page 252](#)
- [xsl:otherwise on page 253](#)
- [xsl:param on page 254](#)
- [xsl:stylesheet on page 255](#)
- [xsl:template on page 256](#)
- [xsl:template match="/" Template on page 257](#)
- [xsl:text on page 259](#)
- [xsl:value-of on page 259](#)
- [xsl:variable on page 260](#)
- [xsl:when on page 261](#)
- [xsl:with-param on page 261](#)

## xsl:apply-templates

---

<b>Syntax</b>	<pre>&lt;xsl:apply-templates select="node-set-expression"&gt;   &lt;xsl:with-param name="qualified-name" select="expression"&gt;     ...   &lt;/xsl:with-param&gt; &lt;/xsl:apply-templates&gt;</pre>
<b>Description</b>	Apply one or more templates, according to the value of the <b>select</b> attribute. If the <b>select</b> attribute is not included, the script recursively processes all child nodes of the current node. If the <b>select</b> attribute is present, the processor only applies templates to the child elements that match the expression of the <b>select</b> attribute, which must evaluate to a node-set. The <b>&lt;xsl:template&gt;</b> instruction dictates which elements are transformed according to which template. The templates that are applied are passed the parameters specified by the <b>&lt;xsl:with-param&gt;</b> elements within the <b>&lt;xsl:apply-templates&gt;</b> instruction.
<b>Attributes</b>	<b>select</b> —(Optional) Selects the nodes to which the processor applies templates. By default, the processor applies templates to the child nodes of the current node.
<b>Usage Examples</b>	See <i>Example: Adding a Final then accept Term to a Firewall</i> and <i>Example: Preventing Import of the Full Routing Table</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">XSLT Templates Overview on page 68</a></li><li>• <a href="#">xsl:call-template on page 248</a></li><li>• <a href="#">xsl:param on page 254</a></li><li>• <a href="#">xsl:template on page 256</a></li><li>• <a href="#">xsl:variable on page 260</a></li><li>• <a href="#">xsl:with-param on page 261</a></li></ul>

## xsl:call-template

---

<b>Syntax</b>	<pre>&lt;xsl:call-template name="qualified-name"&gt;   &lt;xsl:with-param name="qualified-name" select="expression"&gt;     ...   &lt;/xsl:with-param&gt; &lt;/xsl:call-template&gt;</pre>
<b>Description</b>	Call a named template. The <b>&lt;xsl:with-param&gt;</b> elements within the <b>&lt;xsl:call-template&gt;</b> instruction define the parameters that are passed to the template.
<b>Attributes</b>	<b>name</b> —Specifies the name of the template to call.
<b>Usage Examples</b>	See <i>Example: Requiring and Restricting Configuration Statements</i> , <i>Example: Imposing a Minimum MTU Setting</i> , and <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .

- Related Documentation**
- [XSLT Templates Overview on page 68](#)
  - [xsl:apply-templates on page 248](#)
  - [xsl:param on page 254](#)
  - [xsl:template on page 256](#)
  - [xsl:variable on page 260](#)
  - [xsl:with-param on page 261](#)

## xsl:choose

<b>Syntax</b>	<pre> &lt;xsl:choose&gt;   &lt;xsl:when test="boolean-expression"&gt;     ...   &lt;/xsl:when&gt;   &lt;xsl:otherwise&gt;     ...   &lt;/xsl:otherwise&gt; &lt;/xsl:choose&gt; </pre>
<b>Description</b>	<p>Evaluate multiple conditional tests, and execute instructions for the first test that evaluates to TRUE or execute an optional default set of instructions if all tests evaluate to FALSE. The <b>&lt;xsl:choose&gt;</b> instruction contains one or more <b>&lt;xsl:when&gt;</b> elements, each of which tests a Boolean expression. If the test evaluates to TRUE, the XSLT processor executes the instructions in the <b>&lt;xsl:when&gt;</b> element, and ignores all subsequent <b>&lt;xsl:when&gt;</b> elements. The XSLT processor processes only the instructions contained in the first <b>&lt;xsl:when&gt;</b> element whose <b>test</b> attribute evaluates to TRUE. If none of the <b>&lt;xsl:when&gt;</b> elements' <b>test</b> attributes evaluate to TRUE, the content of the optional <b>&lt;xsl:otherwise&gt;</b> element, if one is present, is processed.</p>
<b>Usage Examples</b>	<p>See <i>Example: Configuring Dual Routing Engines</i>, <i>Example: Preventing Import of the Full Routing Table</i>, and <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i>.</p>
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">XSLT Programming Instructions Overview on page 74</a></li> <li>• <a href="#">xsl:for-each on page 251</a></li> <li>• <a href="#">xsl:if on page 252</a></li> <li>• <a href="#">xsl:otherwise on page 253</a></li> <li>• <a href="#">xsl:when on page 261</a></li> </ul>

## xsl:comment

---

<b>Syntax</b>	<pre>&lt;xsl:comment&gt; ... &lt;/xsl:comment&gt;</pre>
<b>Description</b>	<p>Generate a comment node within the final document. The content within the <b>&lt;xsl:comment&gt;</b> element determines the value of the comment. The content must not contain two hyphens next to each other (- -); this sequence is not allowed in comments.</p> <p>XSLT files can contain ordinary comments delimited by <b>&lt;!--</b> and <b>--&gt;</b> such as <b>&lt;!-- ... Insert your comment here ... --&gt;</b>, but these are ignored by the processor. To generate a comment within the final document, use an <b>&lt;xsl:comment&gt;</b> element.</p>
<b>Usage Examples</b>	<p>See <i>Example: Adding a Final then accept Term to a Firewall</i>.</p>
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">xsl:import on page 252</a></li><li>• <a href="#">xsl:stylesheet on page 255</a></li></ul>

## xsl:copy-of

---

<b>Syntax</b>	<pre>&lt;xsl:copy-of select="expression"/&gt;</pre>
<b>Description</b>	<p>Create a copy of what is selected by the expression defined in the <b>select</b> attribute. Namespace nodes, child nodes, and attributes of the current node are automatically copied as well.</p>
<b>Attributes</b>	<p><b>select</b>—XPath expression specifying which nodes to copy.</p>
<b>Usage Examples</b>	<p>See <i>Example: Requiring and Restricting Configuration Statements</i>.</p>
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">xsl:element on page 251</a></li><li>• <a href="#">xsl:text on page 259</a></li><li>• <a href="#">xsl:value-of on page 259</a></li></ul>



## xsl:element

<b>Syntax</b>	<code>&lt;xsl:element name="expression"/&gt;</code>
<b>Description</b>	Create an element node in the output document.
<b>Attributes</b>	<b>name</b> —Specifies the name of the element to be created. The value of the <b>name</b> attribute can be set to an expression that is extracted from the input XML document and evaluated at run time. To do this, enclose an XML element in curly brackets, as in <code>&lt;xsl:element name="{Sisis-level-1}"</code> .
<b>Usage Examples</b>	See <i>Example: Creating a Complex Configuration Based on a Simple Interface Configuration</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">xsl:copy-of on page 250</a></li> <li>• <a href="#">xsl:text on page 259</a></li> <li>• <a href="#">xsl:value-of on page 259</a></li> </ul>

## xsl:for-each

<b>Syntax</b>	<pre> &lt;xsl:for-each select="node-set-expression"&gt; ... &lt;/xsl:for-each&gt; </pre>
<b>Description</b>	Include a looping mechanism that repeats XSL processing for each XML element in the specified node-set. The element nodes are selected by the XPath expression defined by the <b>select</b> attribute. Each of the nodes is then processed according to the instructions contained in the <code>&lt;xsl:for-each&gt;</code> element.
<b>Attributes</b>	<b>select</b> —Specifies an XPath expression that selects the nodes to be processed.
<b>Usage Examples</b>	See <i>Example: Requiring and Restricting Configuration Statements</i> , <i>Example: Imposing a Minimum MTU Setting</i> , <i>Example: Limiting the Number of E1 Interfaces</i> , <i>Example: Adding T1 Interfaces to a RIP Group</i> , <i>Example: Configuring Administrative Groups for LSPs</i> , and <i>Example: Configuring Dual Routing Engines</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">XSLT Programming Instructions Overview on page 74</a></li> <li>• <a href="#">XPath Overview on page 65</a></li> <li>• <a href="#">xsl:choose on page 249</a></li> <li>• <a href="#">xsl:if on page 252</a></li> <li>• <a href="#">xsl:otherwise on page 253</a></li> <li>• <a href="#">xsl:when on page 261</a></li> </ul>

## xsl:if

---

<b>Syntax</b>	<pre>&lt;xsl:if test="expression"&gt; ... &lt;/xsl:if&gt;</pre>
<b>Description</b>	Include a conditional construct that causes instructions to be processed if the expression held in the <b>test</b> attribute evaluates to <b>TRUE</b> .
<b>Attributes</b>	<b>test</b> —Specifies the expression to evaluate.
<b>Usage Examples</b>	<ul style="list-style-type: none"><li>• <i>Example: Adding T1 Interfaces to a RIP Group</i></li><li>• <i>Example: Configuring Dual Routing Engines</i></li><li>• <i>Example: Limiting the Number of E1 Interfaces</i></li><li>• <i>Example: Requiring and Restricting Configuration Statements</i></li></ul>
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">XSLT Programming Instructions Overview on page 74</a></li><li>• <a href="#">xsl:choose on page 249</a></li><li>• <a href="#">xsl:for-each on page 251</a></li><li>• <a href="#">xsl:otherwise on page 253</a></li><li>• <a href="#">xsl:when on page 261</a></li></ul>

## xsl:import

---

<b>Syntax</b>	<pre>&lt;xsl:import href="../../../import/junos.xml"/&gt;</pre>
<b>Description</b>	<p>Import rules from an external style sheet. Provides access to all the declarations and templates within the imported style sheet, and allows you to override them with your own if needed. Any <b>&lt;xsl:import&gt;</b> elements must be the first elements within the style sheet, the first children of the <b>&lt;xsl:stylesheet&gt;</b> document element. The path can be any URI. The <b>../import/junos.xml</b> path shown in the syntax is standard for all commit scripts, op scripts, and event scripts.</p> <p>Imported rules are overwritten by any subsequent matching rules within the importing style sheet. If more than one style sheet is imported, the style sheets imported last override each previous import where the rules match.</p>
<b>Attributes</b>	<b>href</b> —Specifies the location of the imported style sheet.
<b>Usage Examples</b>	<i>See Example: Adding a Final then accept Term to a Firewall, Example: Configuring a Default Encapsulation Type, Example: Configuring Dual Routing Engines, Example: Controlling IS-IS and MPLS Interfaces, Example: Prepending a Global Policy, and Example: Preventing Import of the Full Routing Table.</i>

- Related Documentation**
- [Junos Script Automation: Named Templates in the jcs Namespace Overview on page 40](#)
  - [Junos Script Automation: Global Parameters and Variables in the junos.xml File on page 51](#)
  - *Required Boilerplate for Commit Scripts*
  - *Required Boilerplate for Event Scripts*
  - *Required Boilerplate for Op Scripts*
  - [xsl:stylesheet on page 255](#)

---

## xsl:otherwise

---

<b>Syntax</b>	<pre>&lt;xsl:otherwise&gt; ... &lt;/xsl:otherwise&gt;</pre>
<b>Description</b>	Within an <b>&lt;xsl:choose&gt;</b> instruction, include a default set of instructions that are processed if none of the expressions defined in the <b>test</b> attributes of the <b>&lt;xsl:when&gt;</b> elements evaluate to <b>TRUE</b> .
<b>Usage Examples</b>	See <i>Example: Configuring Dual Routing Engines</i> and <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">XSLT Programming Instructions Overview on page 74</a></li><li>• <a href="#">xsl:choose on page 249</a></li><li>• <a href="#">xsl:for-each on page 251</a></li><li>• <a href="#">xsl:if on page 252</a></li><li>• <a href="#">xsl:when on page 261</a></li></ul>

## xsl:param

---

<b>Syntax</b>	<pre>&lt;xsl:param name="<i>qualified-name</i>" select="<i>expression</i>"&gt; ... &lt;/xsl:param&gt;</pre>
<b>Description</b>	Declare a parameter for a template or for the style sheet as a whole. A template parameter must be declared within the template element. A global parameter, the scope of which is the entire style sheet, must be declared at the top level of the style sheet.
<b>Attributes</b>	<p><b>name</b>—Defines the name of the parameter.</p> <p><b>select</b>—(Optional) XPath expression defining the default value for the parameter, which is used if the person or client application that executes the script does not explicitly provide a value. The <b>select</b> attribute or the content of the <b>&lt;xsl:param&gt;</b> element can define the default value. Do not specify both a <b>select</b> attribute and content; we recommend using the <b>select</b> attribute so as not to create a result tree fragment.</p>
<b>Usage Examples</b>	<ul style="list-style-type: none"><li>• <i>Example: Imposing a Minimum MTU Setting</i></li><li>• <i>Example: Limiting the Number of ATM Virtual Circuits</i></li><li>• <i>Example: Limiting the Number of E1 Interfaces</i></li><li>• <i>Example: Preventing Import of the Full Routing Table</i></li><li>• <i>Example: Requiring and Restricting Configuration Statements</i></li></ul>
<b>Related Documentation</b>	<ul style="list-style-type: none"><li>• <a href="#">XSLT Parameters Overview on page 70</a></li><li>• <a href="#">XSLT Templates Overview on page 68</a></li><li>• <a href="#">xsl:apply-templates on page 248</a></li><li>• <a href="#">xsl:call-template on page 248</a></li><li>• <a href="#">xsl:template on page 256</a></li><li>• <a href="#">xsl:variable on page 260</a></li><li>• <a href="#">xsl:with-param on page 261</a></li></ul>

## xsl:stylesheet

<b>Syntax</b>	<pre>&lt;xsl:stylesheet version="1.0" xmlns:ext="URI"&gt;   &lt;xsl:import href="../../import/junos.xml"/&gt;   ... &lt;/xsl:stylesheet&gt;</pre>
<b>Description</b>	<p>Include the document element for the style sheet. This element defines the root element of the style sheet, which contains all the top-level elements such as global variable and parameter declarations, import elements, and templates. Optionally, namespace mappings, which include an extension prefix and Uniform Resource Identifier (URI), can be included as attributes in the opening <b>&lt;xsl:stylesheet&gt;</b> tag.</p> <p>Any <b>&lt;xsl:import&gt;</b> elements must be the first elements within the style sheet, the first children of the <b>&lt;xsl:stylesheet&gt;</b> document element. The path can be any Uniform Resource Identifier (URI). The <b>../../import/junos.xml</b> path shown in the syntax is standard for all commit scripts, op scripts, and event scripts.</p>
<b>Attributes</b>	<p><b>version</b>—Specifies the version of XSLT that is being used. Junos OS supports XSLT version 1.0.</p> <p><b>xmlns:ext="URI"</b>—(Optional) Maps a namespace prefix to the URI for extension elements.</p>
<b>Usage Examples</b>	<ul style="list-style-type: none"> <li>• <i>Example: Adding a Final then accept Term to a Firewall</i></li> <li>• <i>Example: Configuring Administrative Groups for LSPs</i></li> <li>• <i>Example: Configuring a Default Encapsulation Type</i></li> <li>• <i>Example: Customizing Output of the show interfaces terse Command Using an Op Script</i></li> </ul>
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <i>Required Boilerplate for Commit Scripts</i></li> <li>• <i>Required Boilerplate for Event Scripts</i></li> <li>• <i>Required Boilerplate for Op Scripts</i></li> <li>• <a href="#">XSLT Namespace on page 65</a></li> <li>• <a href="#">xsl:import on page 252</a></li> </ul>

## xsl:template

---

**Syntax**

```
<xsl:template match="pattern" mode="qualified-name" name="qualified-name"
priority="integer">
  <xsl:param name="qualified-name" select="expression">
    ...
  </xsl:param>
  ...
</xsl:stylesheet>
```

**Description** Declare a template that contains rules to apply when a specified node is matched. The **match** attribute associates the template with an XML element. The **match** attribute can also be used to define a template for a whole branch of an XML document. For example, **match="/"** matches the root element of the document. Although the **match** and **name** attributes are optional, one of the two attributes must be included in the template definition.

When templates are applied to a node set using the **<xsl:apply-templates>** instruction, they might be applied in a particular mode; the **mode** attribute in the **<xsl:template>** instruction indicates the mode in which a template needs to be applied for the template to be used. If templates are applied in the specified mode, the **match** attribute is used to determine whether the template can be used with the particular node. If more than one template matches a node in the specified mode, the priority attribute determines which template is used. The highest priority wins. If no priority is specified explicitly, the priority of a template is determined by the **match** attribute.

You can pass template parameters using the **<xsl:with-param>** element. To receive a parameter, the template must contain an **<xsl:param>** element that declares a parameter of that name. These parameters are listed before the body of the template, which is used to process the node and create a result.

**Attributes**

**match**—(Optional) XPath expression specifying the nodes to which to apply the template. If this attribute is omitted, the **name** attribute must be included.

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

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

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

- Usage Examples**
- *Example: Adding a Final then accept Term to a Firewall*
  - *Example: Adding T1 Interfaces to a RIP Group*
  - *Example: Automatically Configuring Logical Interfaces and IP Addresses*
  - *Example: Customizing Output of the show interfaces terse Command Using an Op Script*
  - *Example: Requiring and Restricting Configuration Statements*

- Related Documentation**
- [XSLT Templates Overview on page 68](#)
  - [XSLT Parameters Overview on page 70](#)
  - [xsl:apply-templates on page 248](#)
  - [xsl:call-template on page 248](#)
  - [xsl:param on page 254](#)
  - [xsl:variable on page 260](#)
  - [xsl:with-param on page 261](#)

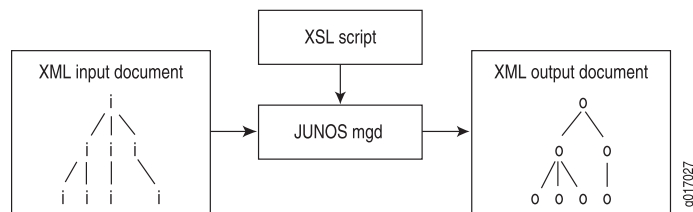
## xsl:template match="/" Template

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

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

Junos OS provides XML-formatted input to a script. Commit script input consists of an XML representation of the post-inheritance candidate configuration file. When you execute a script, the Junos OS management process (mgd) generates an XML-formatted output document as the product of its evaluation of the input document, as shown in [Figure 3 on page 257](#).

**Figure 3: Commit Script Input and Output**



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

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

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

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

```

4   </commit-script-results>
5 </xsl:template>

```

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

```

1 <xsl:template match="/">

```

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

```

2 <commit-script-results>

```

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

```

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

```

Lines 4 and 5 are closing tags.

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

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

```

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

```

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

## Usage Examples

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

```

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

```

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

```

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

```



You can write this:

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

**Related Documentation**

- [Junos Named Templates in the jcs Namespace Summary on page 41](#)
- [Junos Script Automation: Understanding Extension Functions in the jcs and slax Namespaces on page 11](#)
- [Junos Script Automation: Named Templates in the jcs Namespace Overview on page 40](#)
- [Junos Script Automation: Global Parameters and Variables in the junos.xml File on page 51](#)
- [apply-templates on page 195](#)
- [xsl:apply-templates on page 248](#)
- [xsl:template on page 256](#)

## xsl:text

<b>Syntax</b>	<pre>&lt;xsl:text&gt; ... &lt;/xsl:text&gt;</pre>
<b>Description</b>	Insert literal text in the output.
<b>Usage Examples</b>	See <i>Example: Requiring and Restricting Configuration Statements</i> , <i>Example: Imposing a Minimum MTU Setting</i> , <i>Example: Limiting the Number of E1 Interfaces</i> , <i>Example: Controlling IS-IS and MPLS Interfaces</i> , and <i>Example: Adding a Final then accept Term to a Firewall</i> .

## xsl:value-of

<b>Syntax</b>	<pre>&lt;xsl:value-of select="expression"/&gt;</pre>
<b>Description</b>	Extract the value of an XML element and insert it into the output. The <b>select</b> attribute specifies the XPath expression that is evaluated. In the XPath expression, use <b>@</b> to access attributes of elements. Use <b>.</b> to access the contents of the element itself. If the result is a node set, the <b>&lt;xsl:value-of&gt;</b> instruction adds the string value of the first node in that node set; none of the structure of the node is preserved. To preserve the structure of the node, you must use the <b>&lt;xsl:copy-of&gt;</b> instruction instead.
<b>Attributes</b>	<b>select</b> —XPath expression specifying the node or attribute to evaluate.
<b>Usage Examples</b>	<ul style="list-style-type: none"> <li>• <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i></li> <li>• <i>Example: Configuring Administrative Groups for LSPs</i></li> <li>• <i>Example: Controlling IS-IS and MPLS Interfaces</i></li> <li>• <i>Example: Imposing a Minimum MTU Setting</i></li> <li>• <i>Example: Limiting the Number of E1 Interfaces</i></li> </ul>

- Related Documentation
- [xsl:copy-of on page 250](#)

---

## xsl:variable

---

Syntax	<pre>&lt;xsl:variable name="<i>qualified-name</i>" select="<i>expression</i>"&gt; ... &lt;/xsl:variable&gt;</pre>
Description	Declare a local or global variable. If the <b>&lt;xsl:variable&gt;</b> instruction appears at the top level of the style sheet as a child of the <b>&lt;xsl:stylesheet&gt;</b> document element, it is a global variable with a scope that includes the entire style sheet. Otherwise, it is a local variable with a scope of its following siblings and their descendants.
Attributes	<p><b>name</b>—Specifies the name of the variable. After declaration, the variable can be referred to within XPath expressions using this name, prefixed with the \$ character.</p> <p><b>select</b>—(Optional) Determines the value of the variable. The value of the variable is determined either by the <b>select</b> attribute or by the contents of the <b>&lt;xsl:variable&gt;</b> element. Do not specify both a <b>select</b> attribute and some content; we recommend using the <b>select</b> attribute so as not to create a result tree fragment.</p>
Usage Examples	See <i>Example: Limiting the Number of E1 Interfaces</i> , <i>Example: Limiting the Number of ATM Virtual Circuits</i> , <i>Example: Configuring Administrative Groups for LSPs</i> , and <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i> .
Related Documentation	<ul style="list-style-type: none"><li>• <a href="#">XSLT Variables Overview on page 73</a></li><li>• <a href="#">xsl:apply-templates on page 248</a></li><li>• <a href="#">xsl:call-template on page 248</a></li><li>• <a href="#">xsl:param on page 254</a></li><li>• <a href="#">xsl:template on page 256</a></li><li>• <a href="#">xsl:with-param on page 261</a></li></ul>

## xsl:when

<b>Syntax</b>	<pre>&lt;xsl:when test="boolean-expression"&gt; ... &lt;/xsl:when&gt;</pre>
<b>Description</b>	<p>Within an <b>&lt;xsl:choose&gt;</b> instruction, specify a set of processing instructions that are executed when the expression specified in the <b>test</b> attribute evaluates to <b>TRUE</b>. The XSLT processor processes only the instructions contained in the first <b>&lt;xsl:when&gt;</b> element whose <b>test</b> attribute evaluates to <b>TRUE</b>. If none of the <b>&lt;xsl:when&gt;</b> elements' <b>test</b> attributes evaluate to <b>TRUE</b>, the content of the <b>&lt;xsl:otherwise&gt;</b> element, if there is one, is processed.</p>
<b>Attributes</b>	<p><b>test</b>—Specifies a Boolean expression.</p>
<b>Usage Examples</b>	<ul style="list-style-type: none"> <li>• <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i></li> <li>• <i>Example: Configuring Dual Routing Engines</i></li> <li>• <i>Example: Preventing Import of the Full Routing Table</i></li> </ul>
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">XSLT Programming Instructions Overview on page 74</a></li> <li>• <a href="#">xsl:choose on page 249</a></li> <li>• <a href="#">xsl:for-each on page 251</a></li> <li>• <a href="#">xsl:if on page 252</a></li> <li>• <a href="#">xsl:otherwise on page 253</a></li> </ul>

## xsl:with-param

<b>Syntax</b>	<pre>&lt;xsl:with-param name="qualified-name" select="expression"&gt; ... &lt;/xsl:with-param&gt;</pre>
<b>Description</b>	<p>Specify a parameter to pass into a template. This element can be used when applying templates with the <b>&lt;xsl:apply-templates&gt;</b> instruction or when calling templates with the <b>&lt;xsl:call-template&gt;</b> instruction.</p>
<b>Attributes</b>	<p><b>name</b>—Specifies the name of the parameter.</p> <p><b>select</b>—(Optional) XPath expression specifying the value of the parameter. The value of the parameter is determined either by the <b>select</b> attribute or by the contents of the <b>&lt;xsl:with-param&gt;</b> element. Do not specify both a <b>select</b> attribute and content. We recommend using the <b>select</b> attribute to set the parameter so as to prevent the parameter from being passed a result tree fragment as its value.</p>
<b>Usage Examples</b>	<p>See <i>Example: Configuring Dual Routing Engines</i>, <i>Example: Preventing Import of the Full Routing Table</i>, and <i>Example: Automatically Configuring Logical Interfaces and IP Addresses</i>.</p>

- Related Documentation**
- [XSLT Templates Overview on page 68](#)
  - [xsl:apply-templates on page 248](#)
  - [xsl:call-template on page 248](#)
  - [xsl:param on page 254](#)
  - [xsl:template on page 256](#)
  - [xsl:variable on page 260](#)

# Configuration Statements Common to All Scripts

- [load-scripts-from-flash \(Scripts\) on page 263](#)
- [max-datasize on page 264](#)
- [synchronize \(Scripts\) on page 265](#)

## load-scripts-from-flash (Scripts)


<b>Syntax</b>	load-scripts-from-flash;
<b>Hierarchy Level</b>	[edit system scripts]
<b>Release Information</b>	Statement introduced in Junos OS Release 8.5.
<b>Description</b>	Load commit, event, op, and library scripts from the device's flash drive instead of the hard drive. To ensure that the current versions of the scripts are executed, you must manually move the scripts from the hard drive to the flash drive. They are not moved automatically.
<b>Default</b>	If you do not include the <b>load-scripts-from-flash</b> statement, scripts are executed from the device's hard drive.
<b>Required Privilege Level</b>	maintenance—To view this statement in the configuration. maintenance-control—To add this statement to the configuration.
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">Storing Scripts in Flash Memory on page 120</a></li> <li>• <i>scripts</i></li> </ul>

## max-datasize

---

<b>Syntax</b>	<code>max-datasize size;</code>
<b>Hierarchy Level</b>	[edit event-options event-script], [edit system scripts commit], [edit system scripts op]
<b>Release Information</b>	Statement introduced in Junos OS Release 12.3.
<b>Description</b>	Maximum amount of memory allocated for the data segment during execution of a script of the given type. Junos OS sets the maximum memory limit for the executing script to the configured value irrespective of the total memory available on the system at the time of execution. If the executing script exceeds the specified maximum memory limit for that script type, it exits gracefully.
<b>Default</b>	If you do not include the <b>max-datasize</b> statement, the system allocates half of the total available memory of the system up to a maximum value of 128 MB for the data segment portion of the executed script.
<b>Options</b>	<b>size</b> —Maximum amount of memory allocated for the data segment during execution of a script of the given type. If you do not specify a unit of measure, the default is bytes. <b>Syntax:</b> <b>size</b> to specify bytes, <b>sizek</b> to specify KB, <b>sizem</b> to specify MB, or <b>sizeg</b> to specify GB <b>Range:</b> 2,3068,672 (22 MB) through 1,073,741,824 (1 GB)
<b>Required Privilege Level</b>	<b>maintenance</b> —To view this statement in the configuration. <b>maintenance-control</b> —To add this statement to the configuration.
<b>Related Documentation</b>	<ul style="list-style-type: none"><li><i>max-policies</i></li><li><a href="#">Example: Configuring Limits on Executed Event Policies and Memory Allocation for Scripts on page 153</a></li></ul>

## synchronize (Scripts)

<b>Syntax</b>	synchronize;
<b>Hierarchy Level</b>	[edit system scripts]
<b>Release Information</b>	Statement introduced in Junos OS Release 13.2.
<b>Description</b>	<p>Synchronize all commit, event, lib, and op scripts from the requesting Routing Engine to the responding Routing Engine when executing a <b>commit synchronize</b> operation. The device synchronizes all scripts regardless of whether they are enabled in the configuration or have been updated since the last synchronization.</p> <p>When you issue the <b>commit synchronize</b> command, the device performs a commit check on the requesting Routing Engine, synchronizes all scripts to the other Routing Engine, synchronizes, performs a commit check, and commits the configuration on the responding Routing Engine, and finally commits the configuration on the requesting Routing Engine. If the commit check operation fails for the requesting Routing Engine, the process stops, and the scripts are not copied to the responding Routing Engine. If the commit check or commit operation fails for the responding Routing Engine, the scripts are still synchronized since the synchronization occurs prior to the commit check operation on the responding Routing Engine.</p> <p>If the <b>load-scripts-from-flash</b> statement is configured for the requesting Routing Engine, the device synchronizes the scripts from flash memory on the requesting Routing Engine to flash memory on the responding Routing Engine. Otherwise, the device synchronizes the scripts from the hard disk on the requesting Routing Engine to the hard disk on the responding Routing Engine.</p>
<div>  <p><b>NOTE:</b> Configuring the <b>synchronize</b> statement causes the device to synchronize all scripts even if they have not been updated since the last synchronization. If the device has a large number of scripts that are infrequently updated, it might be more suitable to synchronize scripts either manually using the <b>request system scripts synchronize</b> operational mode command or on a per-commit basis using the <b>commit synchronize scripts</b> command.</p> </div>	
<b>Required Privilege Level</b>	<p>maintenance—To view this statement in the configuration.</p> <p>maintenance-control—To add this statement to the configuration.</p>
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">Example: Configuring Script Synchronization Between Routing Engines on page 135</a></li> <li>• <a href="#">Synchronizing Scripts Between Routing Engines on page 269</a></li> <li>• <i>commit</i></li> <li>• <a href="#">request system scripts synchronize on page 281</a></li> </ul>





## PART 3

# Administration

- [Synchronizing Scripts Between Routing Engines on page 269](#)
- [Operational Commands on page 275](#)



# Synchronizing Scripts Between Routing Engines

- [Synchronizing Scripts Between Routing Engines on page 269](#)

## Synchronizing Scripts Between Routing Engines

---

- [Understanding Script Synchronization Between Routing Engines on page 269](#)
- [Script Synchronization Between Routing Engines on page 270](#)

### Understanding Script Synchronization Between Routing Engines

Starting with Junos OS Release 13.2, you can manually synchronize commit, event, lib, and op scripts between Routing Engines on a device running Junos OS or configure the device to automatically synchronize scripts between Routing Engines when you commit and synchronize the configuration. When invoked, the device synchronizes the scripts from the Routing Engine on which you execute the request (the requesting Routing Engine) to the other Routing Engine (the responding Routing Engine).

In operational mode, you can manually synchronize scripts from the requesting Routing Engine to the responding Routing Engine using the **request system scripts synchronize** command. The command permits you to customize the scope of the synchronization. You can synchronize a single script, all scripts of a specific type, or all scripts on the device. You also have the option to synchronize scripts based on filename or on the timestamp of the file.

In configuration mode, you have the option to synchronize all scripts when you commit and synchronize the configuration. To synchronize scripts on a per-commit basis, use the **commit synchronize scripts** command when you commit and synchronize the configuration. Alternatively, you can configure the device to automatically synchronize scripts from the requesting Routing Engine to the responding Routing Engine every time you issue a **commit synchronize** command. To ensure that scripts are copied from the requesting Routing Engine to the responding Routing Engine during a **commit synchronize** operation, configure the **synchronize** statement at the **[edit system scripts]** hierarchy level.

When you synchronize the configuration and scripts, the device:

1. Performs a commit check on the requesting Routing Engine
2. Synchronizes scripts to the responding Routing Engine
3. Synchronizes the configuration to the responding Routing Engine
4. Performs a commit check on the responding Routing Engine
5. Commits the configuration on the responding Routing Engine
6. Commits the configuration on the requesting Routing Engine

This process ensures that any commit scripts that are required for a successful commit operation are present on the responding Routing Engine before committing the configuration. If the commit check operation fails for the requesting Routing Engine, the process stops, and the scripts are not copied to the responding Routing Engine. If the commit check or commit operation fails for the responding Routing Engine, the scripts are still synchronized since the synchronization occurs prior to the commit check operation on the responding Routing Engine.

When synchronizing scripts, the device running Junos OS determines the script source and destination directories based on whether the **load-scripts-from-flash** statement is present in the configuration for each Routing Engine. If the **load-scripts-from-flash** statement is configured for the requesting Routing Engine, the device synchronizes the scripts that are in flash memory. Otherwise, the device synchronizes the scripts that are on the hard disk. If the **load-scripts-from-flash** statement is present in the final configuration for the responding Routing Engine, the scripts are synchronized to flash memory. Otherwise, the scripts are synchronized to the hard disk. The device synchronizes a script regardless of whether it is enabled in the configuration or has been updated since the last synchronization.

The **request system scripts refresh-from** operational mode command permits you to manually refresh a single script from a remote URL. Starting with Junos OS Release 13.2, you can synchronize the updated script to the other Routing Engine by including the **sync** option when you execute the command. When you execute the command, if the **load-scripts-from-flash** statement is configured on the Routing Engine, the script is refreshed in flash memory. Otherwise, the script is refreshed on the hard disk.

## Script Synchronization Between Routing Engines

Starting with Junos OS Release 13.2, you can manually synchronize commit, event, lib, and op scripts between Routing Engines on a device running Junos OS or configure the device to automatically synchronize scripts between Routing Engines when you commit and synchronize the configuration.

If the **load-scripts-from-flash** statement is configured for the requesting Routing Engine, the device synchronizes the scripts that are in flash memory. Otherwise, the device synchronizes the scripts that are on the hard disk. If the **load-scripts-from-flash** statement is present in the final configuration for the responding Routing Engine, the scripts are synchronized to flash memory. Otherwise, the scripts are synchronized to the hard disk. The device synchronizes a script regardless of whether it is enabled in the configuration or has been updated since the last synchronization.

The following sections outline the different methods for synchronizing scripts:

- [Configuring Script Synchronization Between Routing Engines for Commit Synchronize Operations on page 271](#)
- [Synchronizing Scripts Between Routing Engines on a Per-Commit Basis on page 271](#)
- [Synchronizing Scripts Between Routing Engines from Operational Mode on page 272](#)
- [Synchronizing a Script Between Routing Engines After a Refresh on page 273](#)

### Configuring Script Synchronization Between Routing Engines for Commit Synchronize Operations

---

You can configure a device running Junos OS to synchronize all commit, event, lib, and op scripts from the requesting Routing Engine to the other Routing Engine every time you issue the **commit synchronize** command to commit and synchronize the configuration.

To automatically synchronize scripts between Routing Engines during a **commit synchronize** operation:

1. Configure the **synchronize** statement at the **[edit system scripts]** hierarchy level.

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

2. Commit and synchronize the configuration.

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

When you issue the first and subsequent **commit synchronize** commands, the device performs a commit check on the requesting Routing Engine, synchronizes all scripts to the other Routing Engine, synchronizes, performs a commit check, and commits the configuration on the responding Routing Engine, and finally commits the configuration on the requesting Routing Engine. If the commit check operation fails for the requesting Routing Engine, the process stops, and the scripts are not copied to the responding Routing Engine. If the commit check or commit operation fails for the responding Routing Engine, the scripts are still synchronized since the synchronization occurs prior to the commit check operation on the responding Routing Engine.

Configuring the **synchronize** statement causes the device to synchronize all scripts even if they have not been updated since the last synchronization. If the device has a large number of scripts that are infrequently updated, it might be more suitable to synchronize scripts either manually using the **request system scripts synchronize** operational mode command or on a per-commit basis using the **commit synchronize scripts** command.

### Synchronizing Scripts Between Routing Engines on a Per-Commit Basis

---

You can synchronize all commit, event, lib, and op scripts from the requesting Routing Engine to the other Routing Engine on a device running Junos OS on a per-commit basis using the **commit synchronize scripts** command when you commit and synchronize the configuration. This is an alternative to configuring the device to synchronize scripts every time you execute a **commit synchronize** operation.

To synchronize scripts between Routing Engines on a per-commit basis:

1. Make all necessary changes to the configuration.
2. Issue the **commit synchronize scripts** command.

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

When you issue the **commit synchronize scripts** command, the device performs a commit check on the requesting Routing Engine, synchronizes all scripts to the other Routing Engine, synchronizes, performs a commit check, and commits the configuration on the responding Routing Engine, and finally commits the configuration on the requesting Routing Engine. If the commit check operation fails for the requesting Routing Engine, the process stops, and the scripts are not copied to the responding Routing Engine. If the commit check or commit operation fails for the responding Routing Engine, the scripts are still synchronized since the synchronization occurs prior to the commit check operation on the responding Routing Engine.

### Synchronizing Scripts Between Routing Engines from Operational Mode

You can manually synchronize scripts from the requesting Routing Engine to the other Routing Engine on a device running Junos OS by using the **request system scripts synchronize** operational mode command. You can synchronize a single script, all scripts of a specific type, or all scripts on the device. You also have the option to synchronize scripts based on filename or on the timestamp of the file.

To manually synchronize scripts between Routing Engines, issue the **request system scripts synchronize** command with the desired options.

```
user@host> request system scripts synchronize (all | commit | event | lib | op)
<file filename> <newer-than time>
```

Specify **all** to synchronize all scripts present on the requesting Routing Engine to the responding Routing Engine. Specify **commit**, **event**, **lib**, or **op** to synchronize all scripts of the given type to the other Routing Engine. Include the **file** option or the **newer-than** option to narrow the scope to only synchronize scripts with the specified filename or date criteria. The format for the **newer-than** argument is YYYY-MM-DD.HH:MM:SS.

For example, the following command synchronizes all commit, event, lib, and op scripts that have a timestamp newer than 2012-05-15:

```
user@host> request system scripts synchronize all newer-than 2012-05-15
```

The following command synchronizes a single op script with the filename **vpn-info.slax**.

```
user@host> request system scripts synchronize op file vpn-info.slax
```

A synchronize operation might fail if, for example, you request to synchronize a script that does not exist or if the responding Routing Engine cannot handle the request at that time, because it is performing other CPU-intensive operations. If the synchronize operation fails, the device generates an error message.

The following command requests to synchronize a single event script, but the script does not exist in the event scripts directory, so the device issues an error.

```
user@host> request system scripts synchronize event file nonexistent-file.slax  
  
error: Invalid directory: No such file or directory  
warning: No script will be pushed to other RE
```

The following command requests to synchronize a single event script, but the responding Routing Engine does not have the resources to perform the synchronization, so the device issues an error. The device also logs a UI\_SCRIPTS\_COPY\_FAILED error in the system log file with a severity level of error.

```
user@host> request system scripts synchronize event file ospf-neighbor.slax  
  
error: Unable to copy scripts to re: re1
```

### **Synchronizing a Script Between Routing Engines After a Refresh**

---

You can manually refresh a single script from a remote URL and synchronize the updated script to the other Routing Engine on a device running Junos OS by using the **request system scripts refresh-from** operational mode command with the **sync** option.

To manually refresh a single script from a remote source and then synchronize the script to the other Routing Engine, issue the **request system scripts refresh-from** command with the **sync** option, and specify the script type, filename, and remote URL. Specify the URL as an HTTP URL, FTP URL, or secure copy (scp)-style remote file specification.

```
user@host> request system scripts refresh-from (commit | event | op) file filename url url  
sync
```

The system uses the script type to determine the directory on the device where the script resides. If the **load-scripts-from-flash** statement is present in the configuration for the Routing Engine, the system refreshes the script in flash memory. Otherwise, the system refreshes the script on the hard disk.

#### **Related Documentation**

- [Example: Configuring Script Synchronization Between Routing Engines on page 135](#)
- [commit](#)
- [synchronize \(Scripts\) on page 265](#)
- [request system scripts synchronize on page 281](#)
- [request system scripts refresh-from on page 279](#)





## CHAPTER 20

# Operational Commands

- `op invoke-debugger cli`
- `request system scripts convert`
- `request system scripts refresh-from`
- `request system scripts synchronize`

## op invoke-debugger cli

---

Syntax	<b>op invoke-debugger cli</b> <i>filename</i> <detail> < <i>argument-name argument-value</i> >
Release Information	Command introduced in Junos OS Release 13.1.
Description	Invoke the SLAX debugger to trace the execution of an op script that is enabled in the configuration.
Options	<b>detail</b> — Display detailed output.  <b><i>argument-name argument-value</i></b> —(Optional) Specify one or more arguments for the script. For each argument you include on the command line, you must specify a corresponding value for the argument.  <b><i>filename</i></b> —Script filename
Additional Information	For more information about Junos op scripts, see the <i>Junos OS Configuration and Operations Automation Guide</i> .
Required Privilege Level	maintenance
Related Documentation	<ul style="list-style-type: none"><li>• <a href="#">SLAX Debugger, Profiler, and callflow on page 181</a></li></ul>
List of Sample Output	<a href="#">op invoke-debugger cli on page 276</a> <a href="#">op invoke-debugger cli script argument1 value1 on page 276</a>
Output Fields	When you enter the <b>op invoke-debugger cli</b> command, the CLI displays the (sdb) prompt where you can enter SLAX debugger commands.

### Sample Output

#### op invoke-debugger cli

```
user@host> op invoke-debugger cli script1
sdb: The SLAX Debugger (version )
Type 'help' for help
(sdb)
```

#### op invoke-debugger cli script argument1 value1

```
user@host> op invoke-debugger cli script2 interface ge-0/2/0.0
sdb: The SLAX Debugger (version )
Type 'help' for help
(sdb)
```

## request system scripts convert

<b>Syntax</b>	<code>request system scripts convert (slax-to-xslt   xslt-to-slax) source <i>source/filename</i> destination <i>destination/&lt;filename&gt;</i> &lt;partial&gt; &lt;version (1.0   1.1)&gt;</code>
<b>Release Information</b>	Command introduced in Junos OS Release 8.2. Command introduced in Junos OS Release 9.0 for EX Series switches. <b>partial</b> and <b>version</b> options added in Junos OS Release 12.2.
<b>Description</b>	Convert an Extensible Stylesheet Language Transformations (XSLT) script to Stylesheet Language, Alternative syntax (SLAX), or convert a SLAX script to XSLT.
<b>Options</b>	<p><b>destination <i>destination/&lt;filename&gt;</i></b>—Specify a destination for the converted file.</p> <p>Optionally, you can specify a filename for the converted file. If you do not specify a filename, the software assigns one automatically. The default destination filename is <b>SLAX-Conversion-Temp</b> or <b>slax-temp</b> depending on the Junos OS release, with a randomly generated, five-character, alpha-numeric extension. For example, the software converts a source file called <b>test.xml</b> to <b>slax-temp.kWwQk</b>. The software converts a source file called <b>test1.slax</b> to <b>slax-temp.zN61h</b>.</p> <p><b>partial</b>—(Optional) Convert partial script input.</p> <p><b>slax-to-xslt</b>—Convert a SLAX script to XSLT.</p> <p><b>source <i>source/filename</i></b>—Specify a source file that you want to convert.</p> <p><b>version</b>—(Optional) Specify the SLAX version listed in the version statement of the generated script for XSLT-to-SLAX conversions. Acceptable values are 1.0 and 1.1. The default is 1.1.</p> <p><b>xslt-to-slax</b>—Convert an XSLT script to SLAX.</p>
<b>Required Privilege Level</b>	maintenance
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">Converting Scripts Between SLAX and XSLT on page 81</a></li> </ul>
<b>List of Sample Output</b>	<a href="#">request system scripts convert slax-to-xslt on page 277</a> <a href="#">request system scripts convert xslt-to-slax on page 278</a>
<b>Output Fields</b>	When you enter this command, you are provided feedback on the status of your request.

## Sample Output

### request system scripts convert slax-to-xslt

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

#### request system scripts convert xslt-to-slax

```
user@host> request system scripts convert xslt-to-slax source /var/db/scripts/commit/script1.xml  
destination /var/db/scripts/commit partial version 1.0  
conversion complete
```

## request system scripts refresh-from

<b>Syntax</b>	<code>request system scripts refresh-from (commit   event   op) file <i>filename</i> url <i>url</i> &lt;sync&gt;</code>
<b>Release Information</b>	Command introduced in Junos OS Release 10.1. Option <b>sync</b> introduced in Junos OS Release 13.2.
<b>Description</b>	<p>Refresh a script from a remote source at the specified URL. The system uses the script type to determine the directory on the device where the script resides. If you include the <b>sync</b> option, the system refreshes the script on the requesting Routing Engine and then synchronizes the script on the other Routing Engine. If the <b>load-scripts-from-flash</b> statement is present in the configuration for that Routing Engine, the system refreshes the script in flash memory. Otherwise, the system refreshes the script on the hard disk.</p> <p>The Junos XML management protocol equivalent for this operational mode command is:</p> <pre>&lt;request-script-refresh-from&gt;   &lt;type&gt;(commit   op   event)&lt;/type&gt;   &lt;file&gt;<i>filename</i>&lt;/file&gt;   &lt;url&gt;<i>url</i>&lt;/url&gt;   &lt;sync/&gt; &lt;/request-script-refresh-from&gt;</pre>
<b>Options</b>	<p><b>(commit   event   op)</b>—Specify that the script resides in the commit, event, or op script subdirectory under the scripts directory.</p> <p><b>file <i>filename</i></b>—Filename of the script to refresh.</p> <p><b>sync</b>—(Optional) Synchronize the refreshed script from the requesting Routing Engine to the other Routing Engine.</p> <p><b>url <i>url</i></b>—URL of the file that will replace the script provided as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.</p>
<b>Required Privilege Level</b>	maintenance
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">Refreshing a Script from an Alternate Location on page 129</a></li> <li>• <a href="#">Example: Refreshing a Script from an Alternate Source on page 131</a></li> <li>• <a href="#">Example: Configuring and Refreshing from the Master Source for a Script on page 127</a></li> <li>• <a href="#">Synchronizing Scripts Between Routing Engines on page 269</a></li> <li>• <a href="#">request system scripts synchronize on page 281</a></li> </ul>
<b>List of Sample Output</b>	<p><a href="#">request system scripts refresh-from on page 280</a></p> <p><a href="#">request system scripts refresh-from sync on page 280</a></p>

## Sample Output

### request system scripts refresh-from

The following command refreshes the op script **config.slax** from the remote source specified by the URL.

```
user@host> request system scripts refresh-from op file config.slax url
http://host1.juniper.net/config.slax
config.slax                                100%    9    0.0KB/s   00:00
```

The following command refreshes the op script **config1.slax** from the remote source specified by the URL. However, in this case, the script does not exist at the remote site, and the device generates an error message.

```
user@host> request system scripts refresh-from op file config1.slax url
http://host1.juniper.net/config1.slax
refreshing 'config1.slax' from 'http://host1.juniper.net/config1.slax'
fetch-secure: http://host1.juniper.net/config1.slax: Not Found
error: file-fetch failed
error: communication error: rpc failed (file-fetch)
error: error retrieving file http://host1.juniper.net/config1.slax
```

## Sample Output

### request system scripts refresh-from sync

The following command refreshes the op script **config.slax** on the requesting Routing Engine from the remote source specified by the URL. The system then synchronizes the updated script to the other Routing Engine.

```
user@host> request system scripts refresh-from op file config.slax url
http://host1.juniper.net/config.slax sync
config.slax                                100%    9    0.0KB/s   00:00
```

## request system scripts synchronize

<b>Syntax</b>	request system scripts synchronize (all   commit   event   lib   op) <file <i>filename</i> > <newer-than YYYY-MM-DD.HH:MM:SS>
<b>Release Information</b>	Command introduced in Junos OS Release 13.2.
<b>Description</b>	<p>This command is for devices with multiple Routing Engines only. Synchronize scripts from the requesting Routing Engine to the other Routing Engine.</p> <p>Specify <b>all</b> to synchronize all scripts present on the requesting Routing Engine to the responding Routing Engine. Specify <b>commit</b>, <b>event</b>, <b>lib</b>, or <b>op</b> to synchronize all scripts of the given type to the other Routing Engine. Include the <b>file</b> option or the <b>newer-than</b> option to narrow the scope to only synchronize scripts with the specified filename or date criteria.</p> <p>If the <b>load-scripts-from-flash</b> statement is configured for the requesting Routing Engine, the device synchronizes the scripts that are in flash memory. Otherwise, the device synchronizes the scripts that are on the hard disk. If the <b>load-scripts-from-flash</b> statement is configured for the responding Routing Engine, the scripts are synchronized to flash memory. Otherwise, the scripts are synchronized to the hard disk. The device synchronizes a script regardless of whether it is enabled in the configuration.</p> <p>The Junos XML management protocol equivalent for this operational mode command is:</p> <pre>&lt;request-scripts-synchronize&gt;   &lt;all/&gt;   &lt;commit/&gt;   &lt;event/&gt;   &lt;lib/&gt;   &lt;op/&gt;   &lt;file&gt;<i>filename</i>&lt;/file&gt;   &lt;newer-than&gt;YYYY-MM-DD.HH:MM:SS&lt;/newer-than&gt; &lt;/request-scripts-synchronize&gt;</pre>
<b>Options</b>	<p>(all   commit   event   lib   op)—Synchronize all scripts or all scripts of a given type that are present on the requesting Routing Engine to the responding Routing Engine.</p> <p><b>file <i>filename</i></b>—(Optional) Synchronize an individual script with the specified filename from the requesting Routing Engine to the responding Routing Engine. This option can only be used with a specific script type.</p> <p><b>newer-than YYYY-MM-DD.HH:MM:SS</b>—(Optional) Synchronize scripts that are more current than the specified date and time from the requesting Routing Engine to the responding Routing Engine.</p>
<b>Required Privilege Level</b>	maintenance
<b>Related Documentation</b>	<ul style="list-style-type: none"> <li>• <a href="#">Synchronizing Scripts Between Routing Engines on page 269</a></li> <li>• <a href="#">synchronize on page 265</a></li> <li>• <a href="#">request system scripts refresh-from on page 279</a></li> </ul>

**List of Sample Output**    [request system scripts synchronize all on page 282](#)  
[request system scripts synchronize commit file on page 282](#)  
[request system scripts synchronize event file on page 282](#)  
[request system scripts synchronize op newer-than on page 282](#)

## Sample Output

### [request system scripts synchronize all](#)

The following command synchronizes all scripts from the requesting Routing Engine to the responding Routing Engine.

```
user@host> request system scripts synchronize all
user@host>
```

### [request system scripts synchronize commit file](#)

The following command synchronizes the commit script **vpn-commit.slax** from the requesting Routing Engine to the responding Routing Engine. In this case, the script does not exist on the requesting Routing Engine, and the device issues an error message.

```
user@host> request system scripts synchronize commit file vpn-commit.slax
error: Invalid directory: No such file or directory
warning: No script will be pushed to other RE
```

### [request system scripts synchronize event file](#)

The following command synchronizes the event script **ospf-neighbor.slax** from the requesting Routing Engine to the responding Routing Engine. In this case, the responding Routing Engine does not have the resources to perform the operation, and the device issues an error message.

```
user@host> request system scripts synchronize event file ospf-neighbor.slax
error: Unable to copy scripts to re: re1
```

### [request system scripts synchronize op newer-than](#)

The following command synchronizes any op scripts that have been modified after the specified date. In this case, no op scripts meet this requirement.

```
user@host> request system scripts synchronize op newer-than 2012-05-30
warning: No script will be pushed to other RE
```



## PART 4

# Index

- [Index on page 285](#)



# Index

## Symbols

#, comments in configuration statements.....	xix
\$junos-context	
variable.....	52
( ), in syntax descriptions.....	xix
< >, in syntax descriptions.....	xix
[ ], in configuration statements.....	xix
{ }, in configuration statements.....	xix
(pipe), in syntax descriptions.....	xix

## A

append SLAX statement.....	194
apply-imports SLAX statement.....	195
apply-templates SLAX statement.....	195
applying templates	
SLAX.....	88
XSLT.....	68
arguments	
standard.....	54
attribute SLAX statement.....	196
attribute-set SLAX statement.....	197
attributes	
SLAX.....	86

## B

base64-decode() function.....	17
base64-encode() function.....	17
bit extension library.....	172
braces, in configuration statements.....	xix
brackets	
angle, in syntax descriptions.....	xix
square, in configuration statements.....	xix
break-lines() function.....	17

## C

call SLAX statement.....	199
callflow	
overview.....	181
SLAX.....	187
using.....	187

capabilities	
retrieving in a NETCONF session.....	24
close() function.....	18
command output	
RPC, displaying.....	59
comments	
SLAX and XSLT.....	84
comments, in configuration statements.....	xix
commit scripts.....	3
converting.....	277
enabling.....	119
extension functions.....	11, 13
usage guidelines.....	11
input and output illustrated.....	257
master source	
configuring.....	125
updating from.....	125
named templates.....	40
remote sources	
overview.....	123
updating from.....	123
storing.....	119, 120
super-user login class, necessity of.....	119
updating	
from alternate location.....	129
from master source.....	125
concat() XSLT function.....	241
concatenating XPath arguments in SLAX.....	87
configuration changes	
with op scripts.....	47
contains() XSLT function.....	242
context node.....	78
conventions	
text and syntax.....	xviii
converting	
SLAX scripts to XSLT.....	81
XSLT scripts to SLAX.....	81
converting scripts.....	277
copy-node SLAX statement.....	200
copy-of SLAX statement.....	201
count() XSLT function.....	242
cURL extension library.....	174
curl:close.....	177
curl:open.....	177
curl:perform.....	177
curl:set.....	178
curl:single.....	179
curl:close.....	177
curl:open.....	177

curl:perform.....	177
curl:set.....	178
curl:single.....	179
curly braces, in configuration statements.....	xix
customer support.....	xx
contacting JTAC.....	xx

## D

dampen() function.....	18
debugger	
SLAX.....	184
decimal-format SLAX statement.....	201
default extension libraries	
libslax.....	171
display xml command	
usage guidelines.....	59
display xml filter.....	59
display xml rpc command	
usage guidelines.....	59
document type definition See DTD	
document() function.....	19
documentation	
comments on.....	xix
dot node.....	78
downloading the libslax distribution.....	171
DTD	
defined.....	59

## E

element SLAX statement.....	203
elements	
SLAX.....	85
else if SLAX statement.....	204
usage guidelines.....	101
else SLAX statement.....	203
usage guidelines.....	101
empty() function.....	20
evaluate() function.....	21
event scripts.....	3
enabling.....	119
extension functions.....	11, 13
usage guidelines.....	11
master source	
configuring.....	125
updating from.....	125
named templates.....	40
remote sources	
overview.....	123
updating from.....	123

storing.....	119, 120
super-user login class, necessity of.....	119
updating	
from alternate location.....	129
from master source.....	125
examples	
configuring master source.....	127
refreshing from an alternate source.....	131
execute() function.....	21
expr SLAX statement.....	205
expr statement in SLAX.....	87
expressions in SLAX.....	87
extension functions See scripts	
base64-decode().....	17
base64-encode().....	17
break-lines().....	17
close().....	18
dampen().....	18
document().....	19
empty().....	20
evaluate().....	21
execute().....	21
first-of().....	22
get-commands().....	23
get-hello().....	24, 139, 141
get-input().....	25
get-protocol().....	25, 139, 141
get-secret().....	26
hostname().....	27
invoke().....	27
jcs namespace.....	11, 13
open().....	28, 139, 141
output().....	30
parse-ip().....	31
printf().....	32
progress().....	33
regex().....	34
slax namespace.....	11, 13
sleep().....	35
split().....	36
sysctl().....	37
syslog().....	37
trace().....	39
extension libraries	
bit.....	172
cURL.....	174
xutil.....	181

**F**

fallback SLAX statement.....	206
first-of() function.....	22
flash memory	
script storage.....	120
font conventions.....	xviii
for SLAX statement.....	207
for-each SLAX statement.....	208
usage guidelines.....	101
function SLAX statement.....	210
functions	
SLAX.....	91
functions (jcs and slax namespace)	
get-hello().....	141
get-protocol().....	141
open().....	141
functions (jcs and slax namespaces)	
break-lines().....	17
dampen().....	18
empty().....	20
first-of().....	22
get-command().....	23
get-input().....	25
get-secret().....	26
output().....	30
printf().....	32
progress().....	33
regex().....	34
sleep().....	35
split().....	36
sysctl().....	37
syslog().....	37
trace().....	39
functions (jcs namespace)	
close().....	18
execute().....	21
get-hello().....	24, 139
get-protocol().....	25, 139
hostname().....	27
invoke().....	27
open().....	28, 139
parse-ip().....	31
functions (slax namespace)	
base64-decode().....	17
base64-encode().....	17
document().....	19
evaluate().....	21

## functions (XSLT, XPath)

concat().....	241
contains().....	242
count().....	242
last().....	242
name().....	243
not().....	243
position().....	243
starts-with().....	244
string-length().....	244
substring-after().....	245
substring-before().....	245

**G**

get-command() function.....	23
get-hello() function.....	24
get-input() function.....	25
get-protocol() function.....	25
get-secret() function.....	26
global parameters	
junos.xml file.....	51
global variable	
junos-context.....	52
junos.xml file.....	52
grep	
with op scripts.....	46

**H**

hostname() function.....	27
--------------------------	----

**I**

if SLAX statement.....	211
usage guidelines.....	101
import file	
junos.xml.....	40, 51
import SLAX statement.....	212
installing the libslax distribution.....	171
invoke() function.....	27

**J**

## jcs namespace extension functions

break-lines().....	17
close().....	18
dampen().....	18
empty().....	20
execute().....	21
first-of().....	22
get-command().....	23
get-hello().....	24

get-input()	25
get-protocol()	25
get-secret()	26
hostname()	27
invoke()	27
open()	28
output()	30
parse-ip()	31
printf()	32
progress()	33
regex()	34
sleep()	35
split()	36
sysctl()	37
syslog()	37
trace()	39
jcs:break-lines() function	17
jcs:close() function	18
jcs:dampen() function	18
jcs:edit-path template	42
jcs:emit-change template	43
jcs:emit-comment template	45
jcs:empty() function	20
jcs:execute() function	21
jcs:first-of() function	22
jcs:get-command() function	23
jcs:get-hello() function	139, 141
jcs:get-input() function	25
jcs:get-protocol() function	139, 141
jcs:get-secret() function	26
jcs:hostname() function	27
jcs:invoke() function	27
jcs:grep template	46, 47
jcs:open() function	139, 141
jcs:output() function	30
jcs:parse-ip() function	31
jcs:printf() function	32
jcs:progress() function	33
jcs:regex() function	34
jcs:sleep() function	35
jcs:split() function	36
jcs:statement template	50
jcs:sysctl() function	37
jcs:syslog() function	37
jcs:trace() function	39
Junos extension functions	11, 13
Junos named templates	40
Junos OS	
XML	59
Junos XML API	
advantages of	8
overview	7
Junos XML management protocol	
advantages of	8
overview	7
Junos XML management protocol tags	
notational conventions	57
Junos XML protocol	139
Junos XML protocol server	7
Junos XML tags	
displaying CLI output as	59
notational conventions	57
junos-context	
variable	52
junos-netconf session protocol	24, 25, 28, 139
junos.xml file	
global parameters and variables	51
importing	40, 51
parameters	51
templates in	
summaries	40
variable	52
junoscript session protocol	25, 28, 139
<b>K</b>	
key SLAX statement	213
<b>L</b>	
last() XSLT function	242
library	
bit extension	172
curl extension	174
libslax	110
libslax extension	110
script	121
xutil extension	181
libslax	
bit extension library	172
cURL extension library	174
default extension libraries	171
xutil extension library	181
libslax distribution	
downloading	171
installing	171
understanding	109
libslax extension libraries	110
libslax library	110
load-scripts-from-flash statement	120

**M**

manuals	
comments on.....	xix
match SLAX statement.....	215
usage guidelines.....	102
message SLAX statement.....	216
mode SLAX statement.....	216
mvar SLAX statement.....	217

**N**

name() XSLT function.....	243
named templates	
SLAX.....	89
XSLT.....	69
NETCONF protocol.....	139, 141
NETCONF server capabilities.....	24
netconf session protocol.....	24, 25, 28, 139, 141
not() XSLT function.....	243
ns SLAX statement	
usage guidelines.....	103
number SLAX statement.....	218

**O**

op scripts.....	3
changing the configuration.....	47
converting.....	277
enabling.....	119
extension functions.....	11, 13
usage guidelines.....	11
grep.....	46
master source	
specifying.....	125
updating from.....	125
named templates.....	40
remote sources	
overview.....	123
updating from.....	123
storing.....	119, 120
super-user login class, necessity of.....	119
updating	
from alternate location.....	129
from master source.....	125
open() function.....	28
output() function.....	30
output-method SLAX statement.....	222
overview	
SLAX.....	79
XML.....	57
XSLT.....	63

**P**

param SLAX statement.....	225
parameters	
junos.xml file.....	51
SLAX	
declaring.....	93
SLAX,	
passing to functions.....	96
passing to templates.....	94
XSLT.....	70
parentheses, in syntax descriptions.....	xix
parse-ip() function.....	31
position() XSLT function.....	243
preserve-space SLAX statement.....	226
printf() function.....	32
priority SLAX statement.....	226
processing-instruction SLAX statement.....	227
profiler	
SLAX.....	181, 185
programming instructions, XSLT	
<xsl:choose>.....	74
<xsl:for-each>.....	75
<xsl:if>.....	75
progress() function.....	33

**R**

recursion, XSLT.....	77
refresh operation	
commit scripts.....	125
event scripts.....	125
op scripts.....	125
refresh statement	
commit scripts	
usage guidelines.....	125
event scripts	
usage guidelines.....	125
op scripts	
usage guidelines.....	125
refresh-from statement	
commit scripts	
usage guidelines.....	129
event scripts	
usage guidelines.....	129
op scripts	
usage guidelines.....	129
regex() function.....	34
remote source for commit scripts	
overview.....	123
updating from.....	123

remote source for event scripts	
overview.....	123
updating from.....	123
remote source for op scripts	
overview.....	123
updating from.....	123
request system scripts convert.....	277
request system scripts convert command.....	81
request system scripts refresh-from	
command.....	279
request system scripts synchronize command.....	281
result SLAX statement.....	229
RPC	
displaying command output in.....	59

## S

script library.....	121
scripts	
converting.....	277
enabling.....	119
extensions functions.....	11, 13
master source	
configuring.....	125
overview.....	3
storing.....	119
synchronizing on Routing	
Engines.....	135, 265, 269, 270, 281
sdb	
callflow command.....	181
overview.....	181
profile command.....	181
SLAX debugger.....	183
using.....	184
server See Junos XML protocol server	
Service Template Automation	
configuring.....	158
overview.....	157
session protocol	
retrieving.....	25
specifying in automation scripts.....	28, 139, 141
set SLAX statement.....	230
SLAX	
advantages.....	79
applying templates.....	88
attributes.....	86
benefits of.....	80
comments.....	84
converting script to XSLT.....	81
converting to XSLT.....	277

elements.....	85
expr statement.....	87
expressions.....	87
flow of operation illustrated.....	80
functions.....	91
named templates.....	89
operators.....	104
overview.....	79
parameters.....	93
purpose.....	80
statements See SLAX statements	
syntax rules.....	83
using the XSL namespace.....	104
using XSLT elements.....	104
variables.....	97
SLAX callflow command.....	181, 187
SLAX debugger	
callflow command.....	181, 187
cli.....	181
invoking.....	183
libslax.....	181
overview.....	181
profile command.....	181, 185
using.....	184
SLAX extension functions.....	11, 13
slax namespace extension functions	
base64-decode().....	17
base64-encode().....	17
break-lines().....	17
dampen().....	18
document().....	19
empty().....	20
evaluate().....	21
first-of().....	22
get-command().....	23
get-input().....	25
get-secret().....	26
output().....	30
printf().....	32
progress().....	33
regex().....	34
sleep().....	35
split().....	36
sysctl().....	37
syslog().....	37
trace().....	39
SLAX profiler	
overview.....	181
using.....	185



SLAX statements	
append.....	194
apply-imports.....	195
apply-templates.....	195
attribute.....	196
attribute-set.....	197
call.....	199
copy-node.....	200
copy-of.....	201
decimal format.....	201
element.....	203
else.....	203
usage guidelines.....	101
else if.....	204
usage guidelines.....	101
expr.....	205
fallback.....	206
for.....	207
for-each.....	208
usage guidelines.....	101
function.....	210
if.....	211
usage guidelines.....	101
import.....	212
key.....	213
match.....	215
usage guidelines.....	102
message.....	216
mode.....	216
mvar.....	217
ns	
usage guidelines.....	103
number.....	218
output-method.....	222
param.....	225
preserve-space.....	226
priority.....	226
processing-instruction.....	227
result.....	229
set.....	230
sort.....	230
strip-space.....	232
template.....	232
terminate.....	234
trace.....	234
uexpr.....	235
use-attribute-sets.....	236
var.....	237
usage guidelines.....	103
while.....	238
with.....	239
slax:base64-decode() function.....	17
slax:base64-encode() function.....	17
slax:break-lines() function.....	17
slax:dampen() function.....	18
slax:document() function.....	19
slax:empty() function.....	20
slax:evaluate() function.....	21
slax:first-of() function.....	22
slax:get-command() function.....	23
slax:get-input() function.....	25
slax:get-secret() function.....	26
slax:output() function.....	30
slax:printf() function.....	32
slax:progress() function.....	33
slax:regex() function.....	34
slax:sleep() function.....	35
slax:split() function.....	36
slax:sysctl() function.....	37
slax:syslog() function.....	37
slax:trace() function.....	39
slaxproc.....	111
converting script format.....	187
executing SLAX scripts.....	187
formatting SLAX scripts.....	187
mode options.....	187
slaxproc.....	187
using.....	187
validating SLAX scripts.....	187
sleep() function.....	35
sort SLAX statement.....	230
source statement	
commit scripts	
usage guidelines.....	125
event scripts	
usage guidelines.....	125
op scripts	
usage guidelines.....	125
split() function.....	36
standard arguments.....	54
starts-with() XSLT function.....	244
statements in SLAX See SLAX statements	
string-length() XSLT function.....	244
strip-space SLAX statement.....	232
substring-after() XSLT function.....	245
substring-before() XSLT function.....	245

super-user login class	
necessity of for commit scripts.....	119
necessity of for event scripts.....	119
necessity of for op scripts.....	119
support, technical See technical support	
synchronize statement.....	265
synchronizing scripts between Routing Engines.....	135, 269, 270, 281
after refresh.....	273, 279
configuring.....	135, 265, 271
in operational mode.....	272
per-commit basis.....	271
syntax conventions.....	xviii
sysctl() function.....	37
syslog() function.....	37

## T

tags See Junos XML tags, Junos XML management protocol tags	
tags (XML)	
Junos XML.....	57
Junos XML management protocol.....	57
technical support	
contacting JTAC.....	xx
template SLAX statement.....	232
templates See XSLT templates	
applying in SLAX.....	88
jcs:edit-path.....	42
jcs:emit-change.....	43
jcs:emit-comment.....	45
jcs:grep.....	46
jcs:load-configuration.....	47
jcs:statement.....	50
named	
SLAX.....	89
XSLT.....	69
unnamed XSLT.....	68
XSLT.....	68
terminate SLAX statement.....	234
trace SLAX statement.....	234
trace() function.....	39

## U

uexpr SLAX statement.....	235
unnamed XSLT templates.....	68

updating	
commit scripts	
from alternate location.....	129
from master source.....	125
event scripts	
from alternate location.....	129
from master source.....	125
op scripts	
from alternate location.....	129
from master source.....	125
use-attribute-sets SLAX statement.....	236

## V

var SLAX statement.....	237
variable	
junos-context.....	52
junos.xml file.....	52
variables	
SLAX	
declaring.....	97
XSLT.....	73
version SLAX statement.....	237
usage guidelines.....	103

## W

while SLAX statement.....	238
with SLAX statement.....	239

## X

XML	
attributes See Junos XML tags, Junos XML management protocol tags	
namespaces See Junos XML tags, Junos XML management protocol tags	
overview.....	57
tags See Junos XML tags, Junos XML management protocol tags	
XPath	
overview.....	65
XPath functions	
concat().....	241
contains().....	242
count().....	242
last().....	242
name().....	243
not().....	243
position().....	243
starts-with().....	244
string-length().....	244

substring-after().....	245	<xsl:element>.....	251
substring-before().....	245	<xsl:for-each>.....	251
<xsl:apply-templates> XSLT element.....	248	<xsl:if>.....	252
<xsl:call-template> XSLT element.....	248	<xsl:import>.....	252
<xsl:choose> XSLT element.....	249	<xsl:otherwise>.....	253
<xsl:choose> XSLT programming instruction.....	74	<xsl:param>.....	254
<xsl:comment> XSLT element.....	250	<xsl:stylesheet>.....	255
<xsl:copy-of> XSLT element.....	250	<xsl:template>.....	256
<xsl:element> XSLT element.....	251	<xsl:text>.....	259
<xsl:for-each> XSLT element.....	251	<xsl:value-of>.....	259
<xsl:for-each> XSLT programming instruction.....	75	<xsl:variable>.....	260
<xsl:if> XSLT element.....	252	<xsl:when>.....	261
<xsl:if> XSLT programming instruction.....	75	<xsl:with-param>.....	261
<xsl:import> XSLT element.....	252	XSLT functions	
<xsl:otherwise> XSLT element.....	253	concat().....	241
<xsl:param> XSLT element.....	254	contains().....	242
<xsl:stylesheet> XSLT element.....	255	count().....	242
<xsl:template> XSLT element.....	256	last().....	242
<xsl:text> XSLT element.....	259	name().....	243
<xsl:value-of> XSLT element.....	259	not().....	243
<xsl:variable> XSLT element.....	260	position().....	243
<xsl:when> XSLT element.....	261	starts-with().....	244
<xsl:with-param> XSLT element.....	261	string-length().....	244
XSLT		substring-after().....	245
comments.....	84	substring-before().....	245
context node.....	78	XSLT templates	
converting script to SLAX.....	81	summaries.....	40
converting to SLAX.....	277	xutil extension library.....	181
dot node.....	78		
flow of operation illustrated.....	64		
named templates.....	69		
namespace, in SLAX.....	104		
overview.....	63		
parameters.....	70		
programming instructions			
<xsl:choose>.....	74		
<xsl:for-each>.....	75		
<xsl:if>.....	75		
recursion.....	77		
templates.....	40, 68 See XSLT templates		
unnamed templates.....	68		
variables.....	73		
XPath.....	65		
XSLT elements			
<xsl:apply-templates>.....	248		
<xsl:call-template>.....	248		
<xsl:choose>.....	249		
<xsl:comment>.....	250		
<xsl:copy-of>.....	250		

