



Junos[®] OS

Junos XML Management Protocol Developer Guide



Modified: 2018-04-17

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

Juniper Networks, the Juniper Networks logo, Juniper, and Junos are registered trademarks of Juniper Networks, Inc. and/or its affiliates in the United States and other countries. All other trademarks may be 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® OS Junos XML Management Protocol Developer Guide
Copyright © 2018 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 <https://www.juniper.net/support/eula/>. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

Table of Contents

	About the Documentation	xvii
	Documentation and Release Notes	xvii
	Supported Platforms	xvii
	Using the Examples in This Manual	xviii
	Merging a Full Example	xviii
	Merging a Snippet	xix
	Documentation Conventions	xix
	Documentation Feedback	xxi
	Requesting Technical Support	xxii
	Self-Help Online Tools and Resources	xxii
	Opening a Case with JTAC	xxii
Part 1	Overview	
Chapter 1	Junos XML Management Protocol and Junos XML API Overview	3
	Junos XML Management Protocol and Junos XML API Overview	3
	Advantages of Using the Junos XML Management Protocol and Junos XML API	4
	Parsing Device Output	4
	Displaying Device Output	5
Chapter 2	Junos XML Protocol and Junos XML Tags Overview	7
	XML and Junos OS Overview	7
	XML Overview	9
	Tag Elements	9
	Attributes	10
	Namespaces	10
	Document Type Definition	11
	XML and Junos XML Management Protocol Conventions Overview	11
	Request and Response Tag Elements	12
	Child Tag Elements of a Request Tag Element	12
	Child Tag Elements of a Response Tag Element	13
	Spaces, Newline Characters, and Other White Space	13
	XML Comments	14
	XML Processing Instructions	14
	Predefined Entity References	14
	Mapping Junos OS Commands and Command Output to Junos XML Tag Elements	15
	Mapping Command Output to Junos XML Elements	16
	Mapping Commands to Junos XML Request Tag Elements	17
	Mapping for Command Options with Variable Values	17

	Mapping for Fixed-Form Command Options	18
	Mapping Configuration Statements to Junos XML Tag Elements	19
	Mapping for Hierarchy Levels and Container Statements	19
	Mapping for Objects That Have an Identifier	20
	Mapping for Single-Value and Fixed-Form Leaf Statements	21
	Mapping for Leaf Statements with Multiple Values	22
	Mapping for Multiple Options on One or More Lines	23
	Mapping for Comments About Configuration Statements	24
	Using Configuration Response Tag Elements in Junos XML Protocol Requests and Configuration Changes	25
Chapter 3	Junos XML Protocol and JSON Overview	27
	Mapping Junos OS Command Output to JSON Using the CLI	27
	Mapping Junos OS Configuration Statements to JSON	34
	Mapping for Hierarchy Levels and Container Statements	35
	Mapping for Objects That Have an Identifier	36
	Mapping for Single-Value and Fixed-Form Leaf Statements	38
	Mapping for Leaf Statements with Multiple Values	39
	Mapping for Multiple Options on One or More Lines	40
	Mapping for Attributes	42
	Mapping for Configuration Comments	45
Part 2	Managing Junos XML Protocol Sessions	
Chapter 4	Junos XML Protocol Session Overview	51
	Junos XML Protocol Session Overview	51
	Supported Access Protocols for Junos XML Protocol Sessions	52
	Understanding the Client Application's Role in a Junos XML Protocol Session	53
	Understanding the Request Procedure in a Junos XML Protocol Session	54
Chapter 5	Managing Junos XML Protocol Sessions	57
	Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server	57
	Prerequisites for All Access Protocols	58
	Prerequisites for Clear-Text Connections	59
	Prerequisites for SSH Connections	61
	Prerequisites for Outbound SSH Connections	62
	Prerequisites for SSL Connections	65
	Prerequisites for Telnet Connections	67
	Connecting to the Junos XML Protocol Server	68
	Connecting to the Junos XML Protocol Server from the CLI	68
	Connecting to the Junos XML Protocol Server from the Client Application	69
	Starting Junos XML Protocol Sessions	70
	Emitting the <code><?xml?></code> PI	70
	Emitting the Opening <code><junoscript></code> Tag	71
	Parsing the Junos XML Protocol Server's <code><?xml?></code> PI	72
	Parsing the Junos XML Protocol Server's Opening <code><junoscript></code> Tag	73
	Verifying Software Compatibility	74

	Authenticating with the Junos XML Protocol Server for Cleartext or SSL	
	Connections	75
	Submitting an Authentication Request	75
	Interpreting the Authentication Response	76
	Sending Requests to the Junos XML Protocol Server	77
	Operational Requests	78
	Configuration Information Requests	79
	Configuration Change Requests	79
	Parsing the Junos XML Protocol Server Response	80
	Operational Responses	80
	Configuration Information Responses	81
	Configuration Change Responses	82
	Using a Standard API to Parse Response Tag Elements in NETCONF and Junos XML Protocol Sessions	82
	Understanding Character Encoding on Devices Running Junos OS	83
	Handling an Error or Warning in Junos XML Protocol Sessions	84
	Halting a Request in Junos XML Protocol Sessions	85
	Locking and Unlocking the Candidate Configuration or Creating a Private Copy	
	Using the Junos XML Protocol	86
	Locking the Candidate Configuration	86
	Unlocking the Candidate Configuration	87
	Creating a Private Copy of the Configuration	88
	Terminating Junos XML Protocol Sessions	89
	Ending a Junos XML Protocol Session and Closing the Connection	90
	Sample Junos XML Protocol Session	91
	Exchanging Initialization PIs and Tag Elements	91
	Sending an Operational Request	91
	Locking the Configuration	92
	Changing the Configuration	92
	Committing the Configuration	93
	Unlocking the Configuration	93
	Closing the Junos XML Protocol Session	94
Chapter 6	Junos XML Protocol Tracing Operations	95
	NETCONF and Junos XML Protocol Tracing Operations Overview	95
	Example: Tracing NETCONF and Junos XML Protocol Session Operations	96
Chapter 7	Junos XML Protocol Operations	103
	<abort/>	103
	<close-configuration/>	103
	<commit-configuration>	104
	<get-checksum-information>	108
	<get-configuration>	109
	<kill-session>	114
	<load-configuration>	114
	<lock-configuration/>	118
	<open-configuration>	119
	<request-end-session/>	120
	<request-login>	120
	<rpc>	121

	<unlock-configuration/>	122
Chapter 8	Junos XML Protocol Processing Instructions	123
	<?xml?>	123
	<junoscript>	123
Chapter 9	Junos XML Protocol Response Tags	125
	<abort-acknowledgement/>	125
	<authentication-response>	125
	<challenge>	126
	<checksum-information>	127
	<commit-results>	127
	<commit-revision-information>	128
	<database-status>	129
	<database-status-information>	131
	<end-session/>	131
	<load-configuration-results>	132
	<reason>	132
	<routing-engine>	133
	<rpc-reply>	134
	<xnm:error>	134
	<xnm:warning>	136
Chapter 10	Junos XML Element Attributes	139
	active	139
	count	139
	delete	140
	inactive	141
	insert	142
	junos:changed	143
	junos:changed-localtime	144
	junos:changed-seconds	144
	junos:commit-localtime	144
	junos:commit-seconds	145
	junos:commit-user	145
	junos:group	146
	junos:interface-range	146
	junos:key	147
	junos:position	148
	junos:total	148
	matching	149
	protect	150
	recurse	150
	rename	151
	replace	152
	replace-pattern	153
	start	154
	unprotect	154
	xmlns	155

Part 3	Managing Configurations Using the Junos XML Protocol	
Chapter 11	Changing the Configuration Using the Junos XML Protocol	159
	Requesting Configuration Changes Using the Junos XML Protocol	160
	Uploading and Formatting Configuration Data in a Junos XML Protocol Session	162
	Uploading Configuration Data as a File Using the Junos XML Protocol	162
	Uploading Configuration Data as a Data Stream Using the Junos XML Protocol	164
	Defining the Format of Configuration Data to Upload in a Junos XML Protocol Session	166
	Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session	169
	Replacing the Candidate Configuration Using the Junos XML Protocol	169
	Replacing the Candidate Configuration with New Data	170
	Rolling Back to a Previously Committed Configuration	171
	Replacing the Candidate Configuration with a Rescue Configuration	172
	Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol	173
	Creating New Elements in Configuration Data Using the Junos XML Protocol . . .	175
	Merging Elements in Configuration Data Using the Junos XML Protocol	177
	Replacing Elements in Configuration Data Using the Junos XML Protocol	181
	Replacing Only Updated Elements in Configuration Data Using the Junos XML Protocol	184
	Deleting Elements in Configuration Data Using the Junos XML Protocol	186
	Deleting a Hierarchy Level or Container Object	186
	Deleting a Configuration Object That Has an Identifier	188
	Deleting a Single-Value or Fixed-Form Option from a Configuration Object	190
	Deleting Values from a Multivalue Option of a Configuration Object	192
	Renaming Objects In Configuration Data Using the Junos XML Protocol	194
	Reordering Elements In Configuration Data Using the Junos XML Protocol	197
	Protecting or Unprotecting a Configuration Object Using the Junos XML Protocol	200
	Changing a Configuration Element's Activation State Using the Junos XML Protocol	203
	Deactivating a Newly Created Element	204
	Deactivating or Reactivating an Existing Element	205
	Changing a Configuration Element's Activation State Simultaneously with Other Changes Using the Junos XML Protocol	208
	Replacing an Element and Setting Its Activation State	208
	Reordering an Element and Setting Its Activation State	210
	Renaming an Object and Setting Its Activation State	210
	Example: Replacing an Object and Deactivating It	211
	Replacing Patterns in Configuration Data Using the NETCONF or Junos XML Protocol	212
	Replacing Patterns Globally Within the Configuration	213
	Replacing Patterns Within a Hierarchy Level or Container Object That Has No Identifier	214

	Replacing Patterns for a Configuration Object That Has an Identifier	215
Chapter 12	Committing the Configuration on a Device Using the Junos XML Protocol	217
	Verifying Configuration Syntax Using the Junos XML Protocol	217
	Committing the Candidate Configuration Using the Junos XML Protocol	218
	Committing a Private Copy of the Configuration Using the Junos XML Protocol	219
	Committing a Configuration at a Specified Time Using the Junos XML Protocol	221
	Committing the Candidate Configuration Only After Confirmation Using the Junos XML Protocol	223
	Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol	225
	Synchronizing the Candidate Configuration on Both Routing Engines	226
	Synchronizing the Ephemeral Configuration on Both Routing Engines	228
	Forcing a Synchronized Commit Operation	230
	Synchronizing the Candidate Configuration Simultaneously with Other Operations	231
	Logging a Message About a Commit Operation Using the Junos XML Protocol	233
	Viewing the Configuration Revision Identifier for Determining Synchronization Status of Devices with NMS	235
Chapter 13	Using the Ephemeral Configuration Database	237
	Understanding the Ephemeral Configuration Database	237
	Ephemeral Configuration Database Overview	237
	Ephemeral Database Instances	239
	Ephemeral Database Commit Model	240
	Unsupported Configuration Statements in the Ephemeral Configuration Database	242
	Enabling and Configuring Instances of the Ephemeral Configuration Database	244
	Enabling Ephemeral Database Instances	245
	Configuring Ephemeral Database Options	245
	Opening Ephemeral Database Instances	246
	Configuring Ephemeral Database Instances	246
	Displaying Ephemeral Configuration Data in the Junos OS CLI	248
	Committing an Instance of the Ephemeral Configuration Database Using the NETCONF or Junos XML Protocol	249
Part 4	Requesting Operational and Configuration Information Using the Junos XML Protocol	
Chapter 14	Requesting Operational Information Using the Junos XML Protocol	255
	Requesting Operational Information Using the Junos XML Protocol	255
	Specifying the Output Format for Operational Information Requests in a Junos XML Protocol Session	257

Chapter 15	Requesting Configuration Information Using the Junos XML Protocol . .	269
	Requesting Configuration Data Using the Junos XML Protocol	270
	Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session	272
	Specifying the Output Format for Configuration Data in a Junos XML Protocol Session	276
	Requesting Commit-Script-Style XML Configuration Data Using the Junos XML Protocol	281
	Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol	283
	Specifying Whether Configuration Groups and Interface Ranges Are Inherited or Displayed Separately	283
	Displaying the Source Group for Inherited Configuration Group Elements . .	286
	Displaying the Source Interface Range for Inherited Configuration Elements	288
	Examples: Specifying Output Format for Configuration Groups	291
	Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol	295
	Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol	298
	Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session	302
	Requesting the Complete Configuration Using the Junos XML Protocol	302
	Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using the Junos XML Protocol	304
	Requesting All Configuration Objects of a Specific Type Using the Junos XML Protocol	305
	Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol	307
	Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol	310
	Requesting a Single Configuration Object Using the Junos XML Protocol	312
	Requesting Subsets of Configuration Objects Using Regular Expressions	314
	Requesting Multiple Configuration Elements Using the Junos XML Protocol . . .	317
	Retrieving a Previous (Rollback) Configuration Using the Junos XML Protocol . .	319
	Retrieving the Rescue Configuration Using the Junos XML Protocol	321
	Comparing the Active or Candidate Configuration to a Prior Version Using the Junos XML Protocol	324
	Comparing Two Previous (Rollback) Configurations Using the Junos XML Protocol	325
	Requesting an XML Schema for the Configuration Hierarchy Using the Junos XML Protocol	327
	Requesting an XML Schema for the Configuration Hierarchy	327
	Creating the junos.xsd File	328
	Example: Requesting an XML Schema	328

Part 5	Junos XML Protocol Utilities	
Chapter 16	Junos XML Protocol Perl Client	333
	Understanding the Junos XML Protocol Perl Distribution and Sample Scripts . .	333
	Downloading the Junos XML Protocol Perl Client and Prerequisites Package . .	334
	Installing the Junos XML Protocol Perl Client and Prerequisites Package	335
	Verifying Installation and Version of Perl	336
	Extracting the Junos XML Protocol Perl Client and Sample Scripts	336
	Extracting and Installing the Junos XML Protocol Perl Client Prerequisites Package	337
	Installing the Junos XML Protocol Perl Client	338
Chapter 17	Developing Junos XML Protocol Perl Client Applications	341
	Writing Junos XML Protocol Perl Client Applications	342
	Importing Perl Modules and Declaring Constants in Junos XML Protocol Perl Client Applications	342
	Connecting to the Junos XML Protocol Server Using Perl Client Applications . .	344
	Satisfying Protocol Prerequisites	344
	Group Requests	344
	Obtaining and Recording Parameters Required by the JUNOS::Device Object	344
	Obtaining Application-Specific Parameters	345
	Establishing the Connection	346
	Collecting Parameters Interactively in Junos XML Protocol Perl Client Applications	347
	Converting Disallowed Characters in Junos XML Protocol Perl Client Applications	349
	Mapping Junos OS Commands to Perl Methods	351
	Submitting a Request to the Junos XML Protocol Server in Perl Client Applications	352
	Providing Method Options or Attributes	352
	Submitting a Request	354
	Requesting an Inventory of Hardware Components Using Junos XML Protocol Perl Client Applications	356
	Example: Changing the Configuration Using Junos XML Protocol Perl Client Applications	356
	Handling Error Conditions	357
	Locking the Configuration	358
	Reading In and Parsing the Configuration Data	358
	Loading the Configuration Data	359
	Committing the Configuration	360
	Parsing and Formatting Junos XML Protocol Server Responses in Perl Client Applications	361
	Parsing and Formatting an Operational Response	361
	Parsing and Outputting Configuration Data	363
	Closing the Connection to the Junos XML Protocol Server in Perl Client Applications	367

Chapter 18	Developing Junos XML Protocol C Client Applications	369
	Establishing a Junos XML Protocol Session Using C Client Applications	369
	Accessing and Editing Device Configurations Using Junos XML Protocol C Client Applications	370
Part 6	Configuration Statements and Operational Commands	
Chapter 19	Configuration Statements	381
	ephemeral	382
	instance (Ephemeral Database)	384
	traceoptions (NETCONF and Junos XML Protocol)	385
Chapter 20	Operational Commands	387
	show ephemeral-configuration	388

List of Figures

Part 3	Managing Configurations Using the Junos XML Protocol	
Chapter 13	Using the Ephemeral Configuration Database	237
	Figure 1: Ephemeral Database Instances	240

List of Tables

	About the Documentation	xvii
	Table 1: Notice Icons	xx
	Table 2: Text and Syntax Conventions	xx
Part 1	Overview	
Chapter 2	Junos XML Protocol and Junos XML Tags Overview	7
	Table 3: Predefined Entity Reference Substitutions for Tag Content Values	15
	Table 4: Predefined Entity Reference Substitutions for Attribute Values	15
Part 2	Managing Junos XML Protocol Sessions	
Chapter 4	Junos XML Protocol Session Overview	51
	Table 5: Supported Access Protocols and Authentication Mechanisms	52
Chapter 5	Managing Junos XML Protocol Sessions	57
	Table 6: Junos XML Protocol version 1.0 PI and Opening Tag	74
Part 4	Requesting Operational and Configuration Information Using the Junos XML Protocol	
Chapter 15	Requesting Configuration Information Using the Junos XML Protocol . .	269
	Table 7: Regular Expression Operators for the matching Attribute	315

About the Documentation

- Documentation and Release Notes on page xvii
- Supported Platforms on page xvii
- Using the Examples in This Manual on page xviii
- Documentation Conventions on page xix
- Documentation Feedback on page xxi
- Requesting Technical Support on page xxii

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 <https://www.juniper.net/documentation/>.

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 <https://www.juniper.net/books>.

Supported Platforms

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

- ACX Series
- EX Series
- M Series
- MX Series
- PTX Series
- QFX Series
- SRX Series
- T 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 [CLI Explorer](#).

Documentation Conventions

Table 1 on page xx 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 xx defines the text and syntax conventions used in this guide.

Table 2: Text and Syntax Conventions

Convention	Description	Examples
Bold text like this	Represents text that you type.	To enter configuration mode, type the configure command: user@host> configure
Fixed-width text like this	Represents output that appears on the terminal screen.	user@host> show chassis alarms No alarms currently active
<i>Italic text like this</i>	<ul style="list-style-type: none"> Introduces or emphasizes important new terms. Identifies guide names. Identifies RFC and Internet draft titles. 	<ul style="list-style-type: none"> A policy <i>term</i> is a named structure that defines match conditions and actions. <i>Junos OS CLI User Guide</i> RFC 1997, <i>BGP Communities Attribute</i>
<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@# set system domain-name <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">To configure a stub area, include the stub statement at the [edit protocols ospf area area-id] hierarchy level.The console port is labeled CONSOLE.
< > (angle brackets)	Encloses optional keywords or variables.	stub <default-metric <i>metric</i>>;
(pipe symbol)	Indicates a choice between the mutually exclusive keywords or variables on either side of the symbol. The set of choices is often enclosed in parentheses for clarity.	broadcast multicast (<i>string1</i> <i>string2</i> <i>string3</i>)
# (pound sign)	Indicates a comment specified on the same line as the configuration statement to which it applies.	rsvp { # Required for dynamic MPLS only
[] (square brackets)	Encloses a variable for which you can substitute one or more values.	community name members [<i>community-ids</i>]
Indentation and braces ({ })	Identifies a level in the configuration hierarchy.	[edit] routing-options { static { route default { nexthop <i>address</i>; retain; } } }
;(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">In the Logical Interfaces box, select All Interfaces.To cancel the configuration, click Cancel.
> (bold right angle bracket)	Separates levels in a hierarchy of menu selections.	In the configuration editor hierarchy, select Protocols>Ospf .

Documentation Feedback

We encourage you to provide feedback, comments, and suggestions so that we can improve the documentation. You can provide feedback by using either of the following methods:

- Online feedback rating system—On any page of the Juniper Networks TechLibrary site at <https://www.juniper.net/documentation/index.html>, simply click the stars to rate the content, and use the pop-up form to provide us with information about your experience. Alternately, you can use the online feedback form at <https://www.juniper.net/documentation/feedback/>.

- E-mail—Send your comments to techpubs-comments@juniper.net. Include the document or topic name, URL or page number, and software 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 Partner Support Service 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 <https://www.juniper.net/us/en/local/pdf/resource-guides/7100059-en.pdf>.
- Product warranties—For product warranty information, visit <https://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: <https://www.juniper.net/customers/support/>
- Search for known bugs: <https://prsearch.juniper.net/>
- Find product documentation: <https://www.juniper.net/documentation/>
- Find solutions and answer questions using our Knowledge Base: <https://kb.juniper.net/>
- Download the latest versions of software and review release notes: <https://www.juniper.net/customers/csc/software/>
- Search technical bulletins for relevant hardware and software notifications: <https://kb.juniper.net/InfoCenter/>
- Join and participate in the Juniper Networks Community Forum: <https://www.juniper.net/company/communities/>
- Open a case online in the CSC Case Management tool: <https://www.juniper.net/cm/>

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

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 <https://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 <https://www.juniper.net/support/requesting-support.html>.

PART 1

Overview

- [Junos XML Management Protocol and Junos XML API Overview on page 3](#)
- [Junos XML Protocol and Junos XML Tags Overview on page 7](#)
- [Junos XML Protocol and JSON Overview on page 27](#)

CHAPTER 1

Junos XML Management Protocol and Junos XML API Overview

- [Junos XML Management Protocol and Junos XML API Overview on page 3](#)
- [Advantages of Using the Junos XML Management Protocol and Junos XML API on page 4](#)

Junos XML Management Protocol and Junos XML API Overview

The Junos XML Management Protocol is an XML-based protocol that client applications use to request information from and manage the configuration on routing, switching, and security devices running Junos OS. It uses an XML-based data encoding for the configuration data and remote procedure calls. The Junos XML 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 OS configuration statements and operational mode commands. Junos XML configuration tag elements are the content to which the Junos XML protocol operations apply. Junos XML operational tag elements are equivalent in function to operational mode commands in the CLI, which administrators use to retrieve status information for a device.

Client applications request information and change the configuration on a switch, router, or security device by encoding the request with tag elements from the Junos XML management protocol and Junos XML API and sending it to the Junos XML protocol server on the device. The Junos XML protocol server is integrated into 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 protocol and Junos XML API tag elements, and returns the result to the client application. For example, to request information about the status of a device's interfaces, a client application sends the Junos XML API **<get-interface-information>** request. 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 use the Junos XML protocol to build custom end-user interfaces for configuration and information retrieval and display, such as a Web browser-based interface.

- Related Documentation**
- [Advantages of Using the Junos XML Management Protocol and Junos XML API on page 4](#)
 - [XML and Junos OS Overview on page 7](#)
 - [XML Overview on page 9](#)

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 OS operational request, all statements in the Junos OS configuration hierarchy, and basic operations that are equivalent to configuration mode commands. The tag names clearly indicate the function of an element in an operational or configuration request or a configuration statement.

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

Parsing Device Output

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

The formatted ASCII follows:

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

The corresponding XML-tagged version is:

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

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

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

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

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

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

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

Displaying Device Output

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

- Related Documentation**
- [Junos XML Management Protocol and Junos XML API Overview on page 3](#)
 - [XML Overview on page 9](#)

CHAPTER 2

Junos XML Protocol and Junos XML Tags Overview

- [XML and Junos OS Overview on page 7](#)
- [XML Overview on page 9](#)
- [XML and Junos XML Management Protocol Conventions Overview on page 11](#)
- [Mapping Junos OS Commands and Command Output to Junos XML Tag Elements on page 15](#)
- [Mapping Configuration Statements to Junos XML Tag Elements on page 19](#)
- [Using Configuration Response Tag Elements in Junos XML Protocol Requests and Configuration Changes on page 25](#)

XML and Junos OS Overview

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

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

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

To display the configuration or operational mode command output as Junos XML tag elements instead of as the default formatted ASCII, issue the command, and pipe the output to the **display xml** command. Infrastructure tag elements in the response belong to the Junos XML management protocol. The tag elements that describe Junos OS

configuration or operational data belong to the Junos XML API, which defines the Junos OS content that can be retrieved and manipulated by both the Junos XML management protocol and the NETCONF XML management protocol operations. The following example compares the text and XML output for the **show chassis alarms** operational mode command:

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

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

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

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

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

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

Junos OS automation scripts use XML to communicate with the host device. Junos OS provides XML-formatted input to a script. The script processes the input source tree and then returns XML-formatted output to Junos OS. The script type determines the XML input document that is sent to the script as well as the output document that is returned to Junos OS for processing. Commit script input consists of an XML representation of the

post-inheritance candidate configuration file. Event scripts receive an XML document containing the description of the triggering event. All script input documents contain information pertaining to the Junos OS environment, and some scripts receive additional script-specific input that depends on the script type.

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

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. XML 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 9](#)
- [Attributes on page 10](#)
- [Namespaces on page 10](#)
- [Document Type Definition on page 11](#)

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 Juniper Networks documentation to indicate optional parts of Junos OS CLI command strings.

Junos XML 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, event, and SNMP scripts. The **jcs** prefix is then prepended to the **output** function, which is defined in that namespace.

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

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

Document Type Definition

An XML-tagged document or data set is *structured*, because a set of rules specifies the ordering and interrelationships of the items in it. 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 Management Protocol and Junos XML API Overview on page 3](#)
- [XML and Junos OS Overview on page 7](#)

XML and Junos XML Management Protocol Conventions Overview

A client application must comply with XML and Junos XML management protocol conventions. Each request from the client application must be a *well-formed* XML document; that is, it must obey the structural rules defined in the Junos XML protocol and Junos XML document type definitions (DTDs) for the kind of information encoded in the request. The client application must emit tag elements in the required order and only in legal contexts. Compliant applications are easier to maintain in the event of changes to the Junos OS or Junos XML protocol.

Similarly, each response from the Junos XML protocol server constitutes a well-formed XML document (the Junos XML protocol server obeys XML and Junos XML management protocol conventions).

The following sections describe Junos XML management protocol conventions:

- [Request and Response Tag Elements on page 12](#)
- [Child Tag Elements of a Request Tag Element on page 12](#)
- [Child Tag Elements of a Response Tag Element on page 13](#)
- [Spaces, Newline Characters, and Other White Space on page 13](#)

- [XML Comments on page 14](#)
- [XML Processing Instructions on page 14](#)
- [Predefined Entity References on page 14](#)

Request and Response Tag Elements

A *request* tag element is one generated by a client application to request information about a device's current status or configuration, or to change the configuration. A request tag element corresponds to a CLI operational or configuration command. It can occur only within an `<rpc>` tag. For information about the `<rpc>` element, see [“Sending Requests to the Junos XML Protocol Server” on page 77](#).

A *response* tag element represents the Junos XML protocol server's reply to a request tag element and occurs only within an `<rpc-reply>` tag. For information about the `<rpc-reply>` element, see [“Parsing the Junos XML Protocol Server Response” on page 80](#).

The following example represents an exchange in which a client application emits the `<get-interface-information>` request tag with the `<extensive/>` flag and the Junos XML protocol server returns the `<interface-information>` response element.

Client Application

```
<rpc>
  <get-interface-information>
    <extensive/>
  </get-interface-information>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <interface-information xmlns="URL">
    <!-- children of <interface-information> -->
  </interface-information>
</rpc-reply>
```

T1100



NOTE: This example, like all others in this guide, shows each tag element on a separate line, in the tag streams emitted by both the client application and Junos XML protocol server. In practice, a client application does not need to include newline characters between tag elements, because the server automatically discards such white space. For further discussion, see [“Spaces, Newline Characters, and Other White Space” on page 13](#).

For information about the attributes in the opening `rpc-reply` tag, see [“Parsing the Junos XML Protocol Server Response” on page 80](#). For information about the `xmlns` attribute in the opening `<interface-information>` tag, see [“Requesting Operational Information Using the Junos XML Protocol” on page 255](#).

Child Tag Elements of a Request Tag Element

Some request tag elements contain child tag elements. For configuration requests, each child tag element represents a configuration element (hierarchy level or configuration object). For operational requests, each child tag element represents one of the options you provide on the command line when issuing the equivalent CLI command.

Some requests have mandatory child tag elements. To make a request successfully, a client application must emit the mandatory tag elements within the request tag element's opening and closing tags. If any of the children are themselves container tag elements, the opening tag for each must occur before any of the tag elements it contains, and the closing tag must occur before the opening tag for another tag element at its hierarchy level.

In most cases, the client application can emit children that occur at the same level within a container tag element in any order. The important exception is a configuration element that has an *identifier tag element*, which distinguishes the configuration element from other elements of its type. The identifier tag element must be the first child tag element in the container tag element. Most frequently, the identifier tag element specifies the name of the configuration element and is called **<name>**.

Child Tag Elements of a Response Tag Element

The child tag elements of a response tag element represent the individual data items returned by the Junos XML protocol server for a particular request. The children can be either individual tag elements (empty tags or tag element triples) or container tag elements that enclose their own child tag elements. For some container tag elements, the Junos XML protocol server returns the children in alphabetical order. For other elements, the children appear in the order in which they were created in the configuration.

The set of child tag elements that can occur in a response or within a container tag element is subject to change in later releases of the Junos XML API. Client applications must not rely on the presence or absence of a particular tag element in the Junos XML protocol server's output, nor on the ordering of child tag elements within a response tag element. For the most robust operation, include logic in the client application that handles the absence of expected tag elements or the presence of unexpected ones as gracefully as possible.

Spaces, Newline Characters, and Other White Space

As dictated by the XML specification, the Junos XML protocol server ignores white space (spaces, tabs, newline characters, and other characters that represent white space) that occurs between tag elements in the tag stream generated by a client application. Client applications can, but do not need to, include white space between tag elements. However, they must not insert white space within an opening or closing tag. If they include white space in the contents of a tag element that they are submitting as a change to the candidate configuration, the Junos XML protocol server preserves the white space in the configuration database.

In its responses, the Junos XML protocol server includes white space between tag elements to enhance the readability of responses that are saved to a file: it uses newline characters to put each tag element on its own line, and spaces to indent child tag elements to the right compared to their parents. A client application can ignore or discard the white space, particularly if it does not store responses for later review by human users. However, it must not depend on the presence or absence of white space in any particular location when parsing the tag stream.

For more information about white space in XML documents, see the XML specification from the World Wide Web Consortium (W3C), *Extensible Markup Language (XML) 1.0*, at <http://www.w3.org/TR/REC-xml/>.

XML Comments

Client applications and the Junos XML protocol server can insert XML comments at any point between tag elements in the tag stream they generate, but not within tag elements. Client applications must handle comments in output from the Junos XML protocol server gracefully but must not depend on their content. Client applications also cannot use comments to convey information to the Junos XML protocol server, because the server automatically discards any comments it receives.

XML comments are enclosed within the strings `<!--` and `-->`, and cannot contain the string `--` (two hyphens). For more details about comments, see the XML specification at <http://www.w3.org/TR/REC-xml/>.

The following is an example of an XML comment:

```
<!-- This is a comment. Please ignore it. -->
```

XML Processing Instructions

An XML processing instruction (PI) contains information relevant to a particular protocol and has the following form:

```
<?PI-name attributes?>
```

Some PIs emitted during a Junos XML protocol session include information that a client application needs for correct operation. A prominent example is the `<?xml?>` element, which the client application and Junos XML protocol server each emit at the beginning of every Junos XML protocol session to specify which version of XML and which character encoding scheme they are using. For more information, see “[Starting Junos XML Protocol Sessions](#)” on page 70.

The Junos XML protocol server can also emit PIs that the client application does not need to interpret (for example, PIs intended for the CLI). If the client application does not understand a PI, it must treat the PI like a comment instead of exiting or generating an error message.

Predefined Entity References

By XML convention, there are two contexts in which certain characters cannot appear in their regular form:

- In the string that appears between opening and closing tags (the contents of the tag element)
- In the string value assigned to an attribute of an opening tag

When including a disallowed character in either context, client applications must substitute the equivalent *predefined entity reference*, which is a string of characters that represents the disallowed character. Because the Junos XML protocol server uses the same predefined entity references in its response tag elements, the client application

must be able to convert them to actual characters when processing response tag elements.

[Table 3 on page 15](#) summarizes the mapping between disallowed characters and predefined entity references for strings that appear between the opening and closing tags of a tag element.

Table 3: Predefined Entity Reference Substitutions for Tag Content Values

Disallowed Character	Predefined Entity Reference
& (ampersand)	&
> (greater-than sign)	>
< (less-than sign)	<

[Table 4 on page 15](#) summarizes the mapping between disallowed characters and predefined entity references for attribute values.

Table 4: Predefined Entity Reference Substitutions for Attribute Values

Disallowed Character	Predefined Entity Reference
& (ampersand)	&
' (apostrophe)	'
> (greater-than sign)	>
< (less-than sign)	<
" (quotation mark)	"

As an example, suppose that the following string is the value contained by the `<condition>` element:

```
if (a<b && b>c) return "Peer's not responding"
```

The `<condition>` element looks like this (it appears on two lines for legibility only):

```
<condition>if (a&lt;b &amp;&amp; b&gt;c) return "Peer's not \
    responding"</condition>
```

Similarly, if the value for the `<example>` element's **heading** attribute is **Peer's "age" < 40**, the opening tag looks like this:

```
<example heading="Peer&apos;s &quot;age&quot; &lt;&gt; 40">
```

Mapping Junos OS Commands and Command Output to Junos XML Tag Elements

The Junos XML API is an XML representation of Junos OS configuration statements and operational mode commands. It defines an XML equivalent for all statements in the Junos OS configuration hierarchy and many of the commands that you issue in CLI

operational mode. Each operational mode command with a Junos XML counterpart maps to a request tag element and, if necessary, a response tag element.

Request tag elements are used in remote procedure calls (RPCs) within NETCONF and Junos XML protocol sessions to request information from a device running Junos OS. The server returns the response using Junos XML tag elements enclosed within the response tag element. For example, the **show interfaces** command maps to the **<get-interface-information>** request tag, and the server returns the **<interface-information>** response tag.

The following sections outline how to map commands, command options, and command output to Junos XML tag elements.

- [Mapping Command Output to Junos XML Elements on page 16](#)
- [Mapping Commands to Junos XML Request Tag Elements on page 17](#)
- [Mapping for Command Options with Variable Values on page 17](#)
- [Mapping for Fixed-Form Command Options on page 18](#)

Mapping Command Output to Junos XML Elements

On the Junos OS command-line interface (CLI), to display command output as Junos XML tag elements instead of as the default formatted ASCII text, include the **| display xml** option after the command. The tag elements that describe the Junos OS configuration or operational data belong to the Junos XML API, which defines the Junos OS content that can be retrieved and manipulated by NETCONF and Junos XML management protocol operations.

The following example shows the output from the **show chassis hardware** command issued on an M20 router that is running Junos OS Release 9.3 (the opening **<chassis-inventory>** tag appears on two lines only for legibility). This is identical to the server's response for the **<get-chassis-inventory>** RPC request.

```
user@host> show chassis hardware | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.3R1/junos">
  <chassis-inventory \
    xmlns="http://xml.juniper.net/junos/9.3R1/junos-chassis">
    <chassis junos:style="inventory">
      <name>Chassis</name>
      <serial-number>00118</serial-number>
      <description>M20</description>
      <chassis-module>
        <name>Backplane</name>
        <version>REV 06</version>
        <part-number>710-001517</part-number>
        <serial-number>AB5911</serial-number>
      </chassis-module>
      <chassis-module>
        <name>Power Supply A</name>
        <!-- other child tags of <chassis-module> -->
      </chassis-module>
      <!-- other child tags of <chassis> -->
    </chassis>
  </chassis-inventory>
</rpc-reply>
```


Mapping Commands to Junos XML Request Tag Elements

Information about the available command equivalents in the current release of the Junos OS can be found in the *Junos XML API Operational Developer Reference*. For the mapping between commands and Junos XML tag elements, see the *Junos XML API Operational Developer Reference* “Mapping Between Operational Tag Elements, Perl Methods, and CLI Commands” chapter. For detailed information about a specific operation, see the *Junos XML API Operational Developer Reference* “Summary of Operational Request Tags” chapter.

On the Junos OS CLI, you can display the Junos XML request tag elements for any operational mode command that has a Junos XML counterpart. To display the Junos XML RPC request tags for an operational mode command, include the **| display xml rpc** option after the command.

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

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

Mapping for Command Options with Variable Values

Many CLI commands have options that identify the object that the command affects or reports about, distinguishing the object from other objects of the same type. In some cases, the CLI does not precede the identifier with a fixed-form keyword, but XML convention requires that the Junos XML API define a tag element for every option. To learn the names for each identifier (and any other child tag elements) for an operational request tag element, consult the tag element’s entry in the appropriate DTD or in the *Junos XML API Operational Developer Reference*, or issue the command and command option in the CLI and append the **| display xml rpc** option.

The following example shows the Junos XML tag elements for two CLI operational commands that have variable-form options. In the **show interfaces** command, t3-5/1/0:0 is the name of the interface. In the **show bgp neighbor** command, 10.168.1.222 is the IP address for the BGP peer of interest.

CLI Command	JUNOS XML Tags
show interfaces t3-5/1/0:0	<pre><rpc> <get-interface-information> <interface-name>t3-5/1/0:0</interface-name> </get-interface-information> </rpc></pre>
show bgp neighbor 10.168.1.122	<pre><rpc> <get-bgp-neighbor-information> <neighbor-address>10.168.1.122</neighbor-address> </get-bgp-neighbor-information> </rpc></pre>

T1500

You can display the Junos XML RPC tags for a command and its options in the CLI by executing the command and command option and appending | **display xml rpc**.

```
user@host> show interfaces t3-5/1/0:0 | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/16.1R1/junos">
  <rpc>
    <get-interface-information>
      <interface-name>t3-5/1/0:0</interface-name>
    </get-interface-information>
  </rpc>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

Mapping for Fixed-Form Command Options

Some CLI commands include options that have a fixed form, such as the **brief** and **detail** strings, which specify the amount of detail to include in the output. The Junos XML API usually maps such an option to an empty tag whose name matches the option name.

The following example shows the Junos XML tag elements for the **show isis adjacency** command, which has a fixed-form option called **detail**:

CLI Command	JUNOS XML Tags
show isis adjacency detail	<pre><rpc> <get-isis-adjacency-information> <detail/> </get-isis-adjacency-information> </rpc></pre>

T1501

To view the tags in the CLI:

```
user@host> show isis adjacency detail | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/16.1R1/junos">
  <rpc>
    <get-isis-adjacency-information>
      <detail/>
    </get-isis-adjacency-information>
  </rpc>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

- Related Documentation**
- [Mapping Junos OS Commands to Perl Methods on page 351](#)

Mapping Configuration Statements to Junos XML Tag Elements

The Junos XML API defines a tag element for every container and leaf statement in the configuration hierarchy. At the top levels of the configuration hierarchy, there is almost always a one-to-one mapping between tag elements and statements, and most tag names match the configuration statement name. At deeper levels of the hierarchy, the mapping is sometimes less direct, because some CLI notational conventions do not map directly to XML-compliant tagging syntax.



NOTE: For some configuration statements, the notation used when you type the statement at the CLI configuration-mode prompt differs from the notation used in a configuration file. The same Junos XML tag element maps to both notational styles.

The following sections describe the mapping between configuration statements and Junos XML tag elements:

- [Mapping for Hierarchy Levels and Container Statements on page 19](#)
- [Mapping for Objects That Have an Identifier on page 20](#)
- [Mapping for Single-Value and Fixed-Form Leaf Statements on page 21](#)
- [Mapping for Leaf Statements with Multiple Values on page 22](#)
- [Mapping for Multiple Options on One or More Lines on page 23](#)
- [Mapping for Comments About Configuration Statements on page 24](#)

Mapping for Hierarchy Levels and Container Statements

The **<configuration>** element is the top-level Junos XML container element for configuration statements. It corresponds to the **[edit]** hierarchy level in CLI configuration mode. Most statements at the next few levels of the configuration hierarchy are container statements. The Junos XML container tag element that corresponds to a container statement almost always has the same name as the statement.

The following example shows the Junos XML tag elements for two statements at the top level of the configuration hierarchy. Note that a closing brace in a CLI configuration statement corresponds to a closing Junos XML tag.

CLI Configuration Statements	JUNOS XML Tags
system {	<configuration>
login {	<system>
...child statements...	<login>
}	<!-- tags for child statements -->
}	</login>
	</system>
protocols {	<protocols>
ospf {	<ospf>
...child statements...	<!-- tags for child statements -->
}	</ospf>
}	</protocols>
	</configuration>

T1502

Mapping for Objects That Have an Identifier

At some hierarchy levels, the same kind of configuration object can occur multiple times. Each instance of the object has a unique identifier to distinguish it from the other instances. In the CLI notation, the parent statement for such an object consists of a keyword and identifier of the following form:

```
keyword identifier {
... configuration statements for individual characteristics ...
}
```

keyword is a fixed string that indicates the type of object being defined, and **identifier** is the unique name for this instance of the type. In the Junos XML API, the tag element corresponding to the keyword is a container tag element for child tag elements that represent the object's characteristics. The container tag element's name generally matches the **keyword** string.

The Junos XML API differs from the CLI in its treatment of the identifier. Because the Junos XML API does not allow container tag elements to contain both other tag elements and untagged character data such as an identifier name, the identifier must be enclosed in a tag element of its own. Most frequently, identifier tag elements for configuration objects are called **<name>**. Some objects have multiple identifiers, which usually have names other than **<name>**. To verify the name of each identifier tag element for a configuration object, consult the entry for the object in the *Junos XML API Configuration Developer Reference*.



NOTE: The Junos OS reserves the prefix **junos-** for the identifiers of configuration groups defined within the **junos-defaults** configuration group. User-defined identifiers cannot start with the string **junos-**.

Identifier tag elements also constitute an exception to the general XML convention that tag elements at the same level of hierarchy can appear in any order; the identifier tag element always occurs first within the container tag element.

The configuration for most objects that have identifiers includes additional leaf statements, which represent other characteristics of the object. For example, each BGP group configured at the **[edit protocols bgp group]** hierarchy level has an associated name (the identifier) and can have leaf statements for other characteristics such as type, peer

autonomous system (AS) number, and neighbor address. For information about the Junos XML mapping for leaf statements, see [“Mapping for Single-Value and Fixed-Form Leaf Statements” on page 21](#), [“Mapping for Leaf Statements with Multiple Values” on page 22](#), and [“Mapping for Multiple Options on One or More Lines” on page 23](#).

The following example shows the Junos XML tag elements for configuration statements that define two BGP groups called `<name>` and `<name>`. Notice that the Junos XML `<name>` element that encloses the identifier of each group (and the identifier of the neighbor within a group) does not have a counterpart in the CLI statements.

CLI Configuration Statements	JUNOS XML Tags
protocols {	<configuration>
bgp {	<protocols>
group G1 {	<bgp>
type external;	<group>
peer-as 56;	<name>G1</name>
neighbor 10.0.0.1;	<type>external</type>
	<peer-as>56</peer-as>
	<neighbor>
	<name>10.0.0.1</name>
	</neighbor>
}	</group>
group G2 {	<group>
type external;	<name>G2</name>
peer-as 57;	<type>external</type>
neighbor 10.0.10.1;	<peer-as>57</peer-as>
	<neighbor>
	<name>10.0.10.1</name>
	</neighbor>
}	</group>
}	</bgp>
}	</protocols>
	</configuration>

T1503

Mapping for Single-Value and Fixed-Form Leaf Statements

A *leaf statement* is a CLI configuration statement that does not contain any other statements. Most leaf statements define a value for one characteristic of a configuration object and have the following form:

keyword *value*;

In general, the name of the Junos XML tag element corresponding to a leaf statement is the same as the **keyword** string. The string between the opening and closing Junos XML tags is the same as the **value** string.

The following example shows the Junos XML tag elements for two leaf statements that have a keyword and a value: the **message** statement at the **[edit system login]** hierarchy level and the **preference** statement at the **[edit protocols ospf]** hierarchy level.

CLI Configuration Statements

```

system {
  login {
    message "Authorized users only";
    ...other statements under login...
  }
}
protocols {
  ospf {
    preference 15;
    ...other statements under ospf...
  }
}

```

JUNOS XML Tags

```

<configuration>
  <system>
    <login>
      <message>Authorized users only</message>
      <!-- tags for other child statements -->
    </login>
  </system>
  <protocols>
    <ospf>
      <preference>15</preference>
      <!-- tags for other child statements -->
    </ospf>
  </protocols>
</configuration>

```

T1504

Some leaf statements consist of a fixed-form keyword only, without an associated variable-form value. The Junos XML API represents such statements with an empty tag. The following example shows the Junos XML tag elements for the **disable** statement at the **[edit forwarding-options sampling]** hierarchy level.

CLI Configuration Statement

```

forwarding-options {
  sampling {
    disable;
    ...other statements under sampling ...
  }
}

```

JUNOS XML Tags

```

<configuration>
  <forwarding-options>
    <sampling>
      <disable/>
      <!-- tags for other child statements -->
    </sampling>
  </forwarding-options>
</configuration>

```

T1505

Mapping for Leaf Statements with Multiple Values

Some Junos OS leaf statements accept multiple values, which can be either user-defined or drawn from a set of predefined values. CLI notation uses square brackets to enclose all values in a single statement, as in the following:

statement [*value1 value2 value3 ...*];

The Junos XML API instead encloses each value in its own tag element. The following example shows the Junos XML tag elements for a CLI statement with multiple user-defined values. The **import** statement imports two routing policies defined elsewhere in the configuration.

CLI Configuration Statements

```

protocols {
  bgp {
    group 23 {
      import [ policy1 policy2 ];
    }
  }
}

```

JUNOS XML Tags

```

<configuration>
  <protocols>
    <bgp>
      <group>
        <name>23</name>
        <import>policy1</import>
        <import>policy2</import>
      </group>
    </bgp>
  </protocols>
</configuration>

```

T1506

The following example shows the Junos XML tag elements for a CLI statement with multiple predefined values. The **permissions** statement grants three predefined permissions to members of the **user-accounts** login class.

CLI Configuration Statements

```

system {
  login {
    class user-accounts {
      permissions [ configure admin control ];
    }
  }
}

```

JUNOS XML Tags

```

<configuration>
  <system>
    <login>
      <class>
        <name>user-accounts</name>
        <permissions>configure</permissions>
        <permissions>admin</permissions>
        <permissions>control</permissions>
      </class>
    </login>
  </system>
</configuration>

```

T1507

Mapping for Multiple Options on One or More Lines

For some Junos OS configuration objects, the standard CLI syntax places multiple options on a single line, usually for greater legibility and conciseness. In most such cases, the first option identifies the object and does not have a keyword, but later options are paired keywords and values. The Junos XML API encloses each option in its own tag element. Because the first option has no keyword in the CLI statement, the Junos XML API assigns a name to its tag element.

The following example shows the Junos XML tag elements for a CLI configuration statement with multiple options on a single line. The Junos XML API defines a tag element for both options and assigns a name to the tag element for the first option (10.0.0.1), which has no CLI keyword.

CLI Configuration Statements

```

system {
  backup-router 10.0.0.1 destination 10.0.0.2;
}

```

JUNOS XML Tags

```

<configuration>
  <system>
    <backup-router>
      <address>10.0.0.1</address>
      <destination>10.0.0.2</destination>
    </backup-router>
  </system>
</configuration>

```

T1508

The syntax for some configuration objects includes more than one multioption line. Again, the Junos XML API defines a separate tag element for each option. The following example shows Junos XML tag elements for a **traceoptions** statement at the **[edit protocols isis]** hierarchy level. The statement has three child statements, each with multiple options.

CLI Configuration Statements

```

protocols {
  isis {
    traceoptions {
      file trace-file size 3m files 10 world-readable;

      flag route detail;

      flag state receive;

    }
  }
}

```

JUNOS XML Tags

```

<configuration>
  <protocols>
    <isis>
      <traceoptions>
        <file>
          <filename>trace-file</filename>
          <size>3m</size>
          <files>10</files>
          <world-readable/>
        </file>
        <flag>
          <name>route</name>
          <detail/>
        </flag>
        <flag>
          <name>state</name>
          <receive/>
        </flag>
      </traceoptions>
    </isis>
  </protocols>
</configuration>

```

T1509

Mapping for Comments About Configuration Statements

A Junos OS configuration can include comments that describe statements in the configuration. In CLI configuration mode, the **annotate** command specifies the comment to associate with a statement at the current hierarchy level. You can also use a text editor to insert comments directly into a configuration file. For more information, see the *CLI User Guide*.

The Junos XML API encloses comments about configuration statements in the **<junos:comment>** element. (These comments are different from the comments that are enclosed in the strings **<!--** and **-->** and are automatically discarded by the protocol server.)

In the Junos XML API, the **<junos:comment>** element immediately precedes the element for the associated configuration statement. (If the tag element for the associated statement is omitted, the comment is not recorded in the configuration database.) The comment text string can include one of the two delimiters that indicate a comment in the configuration database: either the **#** character before the comment or the paired strings **/*** before the comment and ***/** after it. If the client application does not include the delimiter, the protocol server adds the appropriate one when it adds the comment to the configuration. The protocol server also preserves any white space included in the comment.

The following example shows the Junos XML tag elements that associate comments with two statements in a sample configuration statement. The first comment illustrates how including newline characters in the contents of the **<junos:comment>** element (**/* New backbone area */**) results in the comment appearing on its own line in the configuration file. There are no newline characters in the contents of the second **<junos:comment>** element, so in the configuration file the comment directly follows the associated statement on the same line.

CLI Configuration Statements	JUNOS XML Tags
<pre> protocols { ospf { /* New backbone area */ area 0.0.0.0 { interface so-0/0/0 { # From jnpr1 to jnpr2 hello-interval 5; } } } } </pre>	<pre> <configuration> <protocols> <ospf> <junos:comment> /* New backbone area */ </junos:comment> <area> <name>0.0.0.0</name> <junos:comment> # From jnpr1 to jnpr2</junos:comment> <interface> <name>so-0/0/0</name> <hello-interval>5</hello-interval> </interface> </area> </ospf> </protocols> </configuration> </pre>

T1510

Using Configuration Response Tag Elements in Junos XML Protocol Requests and Configuration Changes

The Junos XML protocol server encloses its response to each configuration request within **<rpc-reply>** and **<configuration>** elements. Enclosing each configuration response within a **<configuration>** element contrasts with how the server encloses each different operational response in a tag named for that type of response—for example, the **<chassis-inventory>** tag for chassis information or the **<interface-information>** tag for interface information.

The Junos XML tag elements within the **<configuration>** element represent configuration hierarchy levels, configuration objects, and object characteristics, always ordered from higher to deeper levels of the hierarchy. When a client application loads a configuration, it can emit the same tag elements in the same order as the Junos XML protocol server uses when returning configuration information. This consistent representation makes handling configuration information more straightforward. For instance, the client application can request the current configuration, store the Junos XML protocol server's response to a local memory buffer, make changes or apply transformations to the buffered data, and submit the altered configuration as a change to the candidate configuration. Because the altered configuration is based on the Junos XML protocol server's response, it is certain to be syntactically correct. For more information about changing routing platform configuration, see [“Requesting Configuration Changes Using the Junos XML Protocol” on page 160](#).

Similarly, when a client application requests information about a configuration element (hierarchy level or configuration object), it uses the same elements that the Junos XML protocol server will return in response. To represent the element, the client application sends a complete stream of elements from the top of the configuration hierarchy (represented by the **<configuration>** tag) down to the requested element. The innermost element, which represents the level or object, is either empty or includes the identifier tag only. The Junos XML protocol server's response includes the same stream of parent tag elements, but the tag element for the requested configuration element contains all the tag elements that represent the element's characteristics or child levels. For more information, see [“Requesting Configuration Data Using the Junos XML Protocol” on page 270](#).

The tag streams emitted by the Junos XML protocol server and by a client application can differ in the use of white space, as described in [“XML and Junos XML Management Protocol Conventions Overview” on page 11](#).

**Related
Documentation**

- [XML and Junos XML Management Protocol Conventions Overview on page 11](#)
- [Mapping Configuration Statements to Junos XML Tag Elements on page 19](#)

CHAPTER 3

Junos XML Protocol and JSON Overview

- [Mapping Junos OS Command Output to JSON Using the CLI on page 27](#)
- [Mapping Junos OS Configuration Statements to JSON on page 34](#)

Mapping Junos OS Command Output to JSON Using the CLI

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

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

The following example executes the **show chassis hardware** command and displays the output in JSON format. The response is identical to the NETCONF or Junos XML protocol server response for the **<get-chassis-inventory format="json">** RPC request.

```
user@host> show chassis hardware | display json
{
  "chassis-inventory" : [
    {
      "attributes" : {
        "xmlns" : "http://xml.juniper.net/junos/16.1R1/junos-chassis"
      },
      "chassis" : [
        {
          "attributes" : {"junos:style" : "inventory"},
          "name" : [
            {
              "data" : "Chassis"
            }
          ],
          "serial-number" : [
            {
              "data" : "serial-number"
            }
          ],
        }
      ],
    }
  ],
}
```

```

        "description" : [
        {
            "data" : "MX80-48T"
        }
        ],
        "chassis-module" : [
        {
            "name" : [
            {
                "data" : "Midplane"
            }
            ],
            "version" : [
            {
                "data" : "REV 11"
            }
            ],
            "part-number" : [
            {
                "data" : "711-031603"
            }
            ],
            "serial-number" : [
            {
                "data" : "serial-number"
            }
            ],
            "description" : [
            {
                "data" : "MX80-48T"
            }
            ],
            "clei-code" : [
            {
                "data" : "CMMAX10BRD"
            }
            ],
            "model-number" : [
            {
                "data" : "CHAS-MX80-48T-S"
            }
            ]
        }
        ],
        /* additional JSON objects */
    ]
}
]
}

```

Starting in Junos OS Release 16.1, Junos OS uses a new default implementation for serialization for configuration data emitted in JSON format. The new default is as defined in Internet drafts draft-ietf-netmod-yang-json-09, [JSON Encoding of Data Modeled with YANG](#), and draft-ietf-netmod-yang-metadata-06, [Defining and Using Metadata with YANG](#). When displaying the Junos OS configuration in JSON, you can specify the default implementation of the serialization used to display the output. To define the default, configure the desired value at the **[edit system export-format json]** hierarchy level.



NOTE: Starting in Junos OS Release 17.3R1, OpenConfig supports the operational state emitted by daemons directly in JSON format in addition to XML format. To configure JSON compact format, specify the following CLI command:

set system export-format state-data json compact.

This CLI command converts XML format to compact JSON format. Else, it emits the JSON in non-compact format.

The following example executes the **show system uptime** command and displays the output in non-compact and compact JSON format.

```
user@host> show system uptime | display json
```

Non-compact JSON format:

```
{
  "system-uptime-information" : [
    { "attributes" : { "xmlns" : "http://xml.juniper.net/junos/17.4I0/junos" },
      "current-time" : [
        {
          "date-time" : [
            {
              "data" : "2017-07-14 15:10:24 EDT",
              "attributes" : { "junos:seconds" : "1500059424" }
            }
          ]
        }
      ],
      "time-source" : [
        {
          "data" : " LOCAL CLOCK "
        }
      ],
      "system-booted-time" : [
        {
          "date-time" : [
```

```
{
  "data" : "2017-07-09 21:46:09 EDT",
  "attributes" : {"junos:seconds" : "1499651169"}
},
"time-length" : [
  {
    "data" : "4d 17:24",
    "attributes" : {"junos:seconds" : "408255"}
  }
],
"protocols-started-time" : [
  {
    "date-time" : [
      {
        "data" : "2017-07-09 21:47:21 EDT",
        "attributes" : {"junos:seconds" : "1499651241"}
      }
    ],
    "time-length" : [
      {
        "data" : "4d 17:23",
        "attributes" : {"junos:seconds" : "408183"}
      }
    ]
  }
],
"last-configured-time" : [
  {
```

```
"date-time" : [  
  {  
    "data" : "2017-07-14 15:10:19 EDT",  
    "attributes" : {"junos:seconds" : "1500059419"}  
  },  
  {  
    "data" : "00:00:05",  
    "attributes" : {"junos:seconds" : "5"}  
  },  
  {  
    "data" : "regress"  
  }  
],  
"uptime-information" : [  
  {  
    "date-time" : [  
      {  
        "data" : "3:10PM",  
        "attributes" : {"junos:seconds" : "1500059424"}  
      }  
    ],  
    "up-time" : [  
      {  
        "data" : "4 days, 17:24",
```

```
"attributes" : {"junos:seconds" : "408285"}
}
],
"active-user-count" : [
{
"data" : "1",
"attributes" : {"junos:format" : "1 user"}
}
],
"load-average-1" : [
{
"data" : "0.11"
}
],
"load-average-5" : [
{
"data" : "0.08"
}
],
"load-average-15" : [
{
"data" : "0.12"
}
]
}
]
```

Compact JSON format for the same command:


```
{
  "system-uptime-information" :
  {
    "current-time" :
    {
      "date-time" : "2017-07-14 15:09:50 EDT"
    },
    "time-source" : " LOCAL CLOCK ",
    "system-booted-time" :
    {
      "date-time" : "2017-07-09 21:46:09 EDT",
      "time-length" : "4d 17:23"
    },
    "protocols-started-time" :
    {
      "date-time" : "2017-07-09 21:47:21 EDT",
      "time-length" : "4d 17:22"
    },
    "last-configured-time" :
    {
      "date-time" : "2017-07-11 23:50:34 EDT",
      "time-length" : "2d 15:19",
      "user" : "regress"
    },
    "uptime-information" :
    {
      "date-time" : "3:09PM",
      "up-time" : "4 days, 17:24",
      "active-user-count" : "1",
      "load-average-1" : "0.00",
      "load-average-5" : "0.05",
```

```
"load-average-15" : "0.11"
}
}
}
```

Release History Table

Release	Description
17.3R1	Starting in Junos OS Release 17.3R1, OpenConfig supports the operational state emitted by daemons directly in JSON format in addition to XML format. To configure JSON compact format, specify the following CLI command: set system export-format state-data json compact. This CLI command converts XML format to compact JSON format. Else, it emits the JSON in non-compact format.
16.1	Starting in Junos OS Release 16.1, Junos OS uses a new default implementation for serialization for configuration data emitted in JSON format.
14.2	Starting in Junos OS Release 14.2, devices running Junos OS also support a JavaScript Object Notation (JSON) representation of the operational command output and the Junos OS configuration hierarchy.

Related Documentation

- [Mapping Junos OS Commands and Command Output to Junos XML Tag Elements on page 15](#)

Mapping Junos OS Configuration Statements to JSON

A configuration for a device running Junos OS is stored as a hierarchy of statements. The configuration statement hierarchy has two types of statements: *container statements*, which are statements that contain other statements, and *leaf statements*, which do not contain other statements. All of the container and leaf statements together form the *configuration hierarchy*.

The configuration hierarchy can be represented using JavaScript Object Notation (JSON) in addition to formatted ASCII text, Junos XML elements, and configuration mode **set** commands. Starting in Junos OS Release 14.2, you can view the configuration of a device running Junos OS in JSON format by executing the **show configuration | display json** command in the CLI. In addition, starting in Junos OS Release 16.1, you can load JSON-formatted configuration data on the device.



NOTE: Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.



NOTE: Starting in Junos OS Releases 16.1R4, 16.2R2, and 17.1R1, integers in Junos OS configuration data emitted in JSON format are not enclosed in quotation marks. In earlier releases, integers in JSON configuration data were treated as strings and enclosed in quotation marks.

The following sections describe the mapping between the formatted ASCII text and the default format used for Junos OS configuration statements in JSON:

- [Mapping for Hierarchy Levels and Container Statements on page 35](#)
- [Mapping for Objects That Have an Identifier on page 36](#)
- [Mapping for Single-Value and Fixed-Form Leaf Statements on page 38](#)
- [Mapping for Leaf Statements with Multiple Values on page 39](#)
- [Mapping for Multiple Options on One or More Lines on page 40](#)
- [Mapping for Attributes on page 42](#)
- [Mapping for Configuration Comments on page 45](#)

Mapping for Hierarchy Levels and Container Statements

The Junos OS configuration hierarchy is represented in JSON by a JSON object with a single top-level member, or name/value pair, that has the field name set to "**configuration**" and a value that contains a JSON object representing the entire configuration. The **configuration** member corresponds to the **[edit]** hierarchy level in CLI configuration mode. Most statements at the next few levels of the configuration hierarchy are container statements.

In JSON, each Junos OS hierarchy level or container statement is a member of its parent object. The member, or name/value pair, has a field name corresponding to the name of the hierarchy or container. Its value is a JSON object that contains members representing the child containers and leaf statements at that hierarchy level. The object might also contain a member that holds the list of attributes, if any, associated with that hierarchy.

The following example shows the mapping between formatted ASCII text and JSON for two statements at the top level of the configuration hierarchy:

CLI Configuration Statements

```
system {
  login {
    ...child statements...
  }
}
protocols {
  ospf {
    ...child statements...
  }
}
```

JSON Syntax

```
{
  "configuration" : {
    "system" : {
      "login" : {
        ...JSON configuration data...
      }
    }
    "protocols" : {
      "ospf" : {
        ...JSON configuration data...
      }
    }
  }
}
```

Mapping for Objects That Have an Identifier

At some hierarchy levels, the same kind of configuration object can occur multiple times. Each instance of the object has a unique identifier to distinguish it from the other instances. In the CLI notation, the parent statement for such an object might consist of a keyword and identifier or just an identifier.

```
keyword identifier {
... configuration statements for individual characteristics ...
}
```

keyword is a fixed string that indicates the type of object being defined, and **identifier** is a unique name for an instance of that type. In the following example, **user** is a keyword, and **username** is the identifier.

```
user username {
  /* child statements */
}
```

In JSON, all instances of a configuration object are contained within a single name/value pair in which the field name generally matches the **keyword** string, and the value is an array of JSON objects, each of which is an instance of the configuration object. The JSON syntax differs from the CLI in its treatment of the identifier. In JSON, each instance of the configuration object uses a name/value pair for the identifier, where the field name distinguishes this data as the identifier, and the value is the actual unique identifier for the object. Most frequently, the field name is just **name**. Some objects have multiple identifiers, and might use a field name other than **name**. JSON data that specifies an identifier is always listed first within the corresponding object, but after any attribute list included for that object.

```
{
  "keyword" : [
    {
      "@" : {
        "comment" : "comment"
      },
      "name" : "identifier",
      JSON data for individual characteristics
    },
  ],
}
```

```

{
  "name" : "identifier",
  JSON data for individual characteristics
}
]
}

```



NOTE: Junos OS reserves the prefix `junos-` for the identifiers of configuration groups defined within the `junos-defaults` configuration group. User-defined identifiers cannot start with the string `junos-`.

The configuration for most objects that have identifiers includes additional leaf statements, which represent other characteristics of the object. For example, each BGP group configured at the **[edit protocols bgp group]** hierarchy level has an associated name (the identifier) and can have leaf statements for other characteristics such as type, peer autonomous system (AS) number, and neighbor address. For information about the JSON mapping for leaf statements, see [“Mapping for Single-Value and Fixed-Form Leaf Statements” on page 38](#).

The following example shows the mapping of formatted ASCII text to JSON for configuration statements that define two BGP groups named G1 and G2. In the JSON syntax, the **group** member's value is an array that contains a separate JSON object for each BGP group.

CLI Configuration Statements

```

protocols {
  bgp {
    group G1 {
      type external;
      peer-as 64501;
      neighbor 10.0.0.1;
    }
    group G2 {
      type external;
      peer-as 64502;
      neighbor 10.0.10.1;
    }
  }
}

```

JSON Syntax

```

{
  "configuration" : {
    "protocols" : {
      "bgp" : {
        "group" : [
          {
            "name" : "G1",
            "type" : "external",
            "peer-as" : "64501",
            "neighbor" : [
              {
                "name" : "10.0.0.1"
              }
            ]
          }
        ]
      }
    }
  }
}

```

```
    }
  ]
},
{
  "name" : "G2",
  "type" : "external",
  "peer-as" : "64502",
  "neighbor" : [
    {
      "name" : "10.0.10.1"
    }
  ]
}
]
}
}
```

Mapping for Single-Value and Fixed-Form Leaf Statements

A *leaf statement* is a CLI configuration statement that does not contain any other statements. Most leaf statements define a value for one characteristic of a configuration object and have the following form:

keyword *value*;

Junos OS leaf statements are mapped to name/value pairs in JSON. In general, the field name is the same as the **keyword** string, and the value is the same as the **value** string.

The following example shows the JSON mapping for two leaf statements that have a keyword and a value: the **message** statement at the **[edit system login]** hierarchy level and the **preference** statement at the **[edit protocols ospf]** hierarchy level.

CLI Configuration Statements

```
system {
  login {
    message "Authorized users only.";
    ... other statements under login ...
  }
}
protocols {
  ospf {
    preference 15;
    ... other statements under ospf ...
  }
}
```

JSON Syntax

```
{
  "configuration" : {
    "system" : {
      "login" : {
        "message" : "Authorized users only.",
        ... JSON data for other statements under login ...
      }
    },
  },
}
```

```

    "protocols" : {
      "ospf" : {
        "preference" : "15",
        ... JSON data for other statements under ospf ...
      }
    }
  }
}

```

Some leaf statements consist of a fixed-form keyword only, without an associated variable-form value. Junos OS represents such statements in JSON by setting the value in the name/value pair to **[null]**. The following example shows the JSON mapping for the **disable** statement at the **[edit forwarding-options sampling]** hierarchy level.

CLI Configuration Statements

```

forwarding-options {
  sampling {
    disable;
  }
}

```

JSON Syntax

```

{
  "configuration" : {
    "forwarding-options" : {
      "sampling" : {
        "disable" : [null]
      }
    }
  }
}

```

Mapping for Leaf Statements with Multiple Values

Some Junos OS leaf statements accept multiple values, which can be either user-defined or drawn from a set of predefined values. CLI notation uses square brackets to enclose all values in a single statement, as in the following example:

```
keyword [ value1 value2 value3 ...];
```

As discussed in [“Mapping for Single-Value and Fixed-Form Leaf Statements” on page 38](#), leaf statements are mapped to name/value pairs in JSON, where the field name is the same as the **keyword** string. To represent multiple values, the value portion of the JSON data uses an array of comma-separated strings.

The following example shows the JSON mapping for a CLI statement with multiple user-defined values. The **import** statement imports two routing policies defined elsewhere in the configuration. The formatted ASCII text uses a space-separated list of values, whereas the JSON data uses an array with a comma-separated list of strings.

CLI Configuration Statements

```

protocols {
  bgp {

```

```
        group 23 {
            import [ policy1 policy2 ];
        }
    }
}
```

JSON Syntax

```
{
  "configuration" : {
    "protocols" : {
      "bgp" : {
        "group" : [
          {
            "name" : "23",
            "import" : ["policy1", "policy2"]
          }
        ]
      }
    }
  }
}
```

The following example shows the JSON mapping for a CLI statement with multiple predefined values. The **permissions** statement grants three predefined permissions to members of the **user-accounts** login class.

CLI Configuration Statements

```
system {
  login {
    class user-accounts {
      permissions [ admin configure control ];
    }
  }
}
```

JSON Syntax

```
{
  "configuration" : {
    "system" : {
      "login" : {
        "class" : [
          {
            "name" : "user-accounts",
            "permissions" : ["admin", "configure", "control"]
          }
        ]
      }
    }
  }
}
```

Mapping for Multiple Options on One or More Lines

For some Junos OS configuration objects, the standard CLI syntax places multiple options on a single line, usually for greater legibility and conciseness. In most such cases, the first

option identifies the object and does not have a keyword, but later options are paired keywords and values.

In JSON, the same configuration object maps to a name/value pair. The field name is the same as the object name, and the value is a JSON object containing the options, which are represented by name/value pairs. If the first option has no keyword in the CLI statement, the JSON mapping assigns a name, which is equivalent to the option name defined in the schema and used for the corresponding Junos XML tag name.

The following example shows the JSON mapping for a CLI configuration statement with multiple options on a single line. The JSON representation of the **[edit system backup-router]** statement uses name/value pairs for both options and assigns the field name **address** for the first option (10.0.0.1), which has no CLI keyword.

CLI Configuration Statements

```
system {
  backup-router 10.0.0.1 destination 10.0.0.2/32;
}
```

JSON Syntax

```
{
  "configuration" : {
    "system" : {
      "backup-router" : {
        "address" : "10.0.0.1",
        "destination" : ["10.0.0.2/32"]
      }
    }
  }
}
```

The syntax for some configuration objects includes more than one multi-option line. In JSON, the configuration object maps to a name/value pair, as in the previous case. The field name is the same as the object name, and the value is a JSON object containing the options, which are represented by name/value pairs. For each option, the field name is the same as the option name, and the value is a JSON data structure that appropriately represents the configuration data for that option. When an option uses the same keyword but spans multiple lines, the JSON representation combines the options into a single name/value pair. In this case, the value is an array of JSON objects in which each option is mapped to a separate object.

The following example shows the JSON mapping for the **traceoptions** statement at the **[edit protocols isis]** hierarchy level. The **traceoptions** statement has three child statements, each with multiple options. The CLI notation displays the individual **flag** options on separate lines, but the JSON representation combines the **flag** details into a single name/value pair. The value is an array of objects where each object contains the details for one flag.

CLI Configuration Statements

```
protocols {
  isis {
```

```
        traceoptions {
            file trace-file size 3m files 10 world-readable;
            flag route detail;
            flag state receive;
        }
    }
}
```

JSON Syntax

```
{
  "configuration" : {
    "protocols" : {
      "isis" : {
        "traceoptions" : {
          "file" : {
            "filename" : "isis-trace-file",
            "size" : "3m",
            "files" : 10,
            "world-readable" : [null]
          },
          "flag" : [
            {
              "name" : "route",
              "detail" : [null]
            },
            {
              "name" : "state",
              "receive" : [null]
            }
          ]
        }
      }
    }
  }
}
```

Mapping for Attributes

The Junos OS configuration hierarchy can contain tags that modify a hierarchy or statement. For example, if you issue the **deactivate** command to deactivate a statement in the configuration, the **inactive:** tag is prepended to the item in the configuration to indicate this property. The Junos XML API represents this property as an attribute in the opening tag of the XML element.

The JSON representation of the Junos OS configuration uses metadata annotations to represent these properties. The metadata annotations are encoded as members of a single JSON object and include the "@" symbol as or within the name.

The metadata object representing the attribute list for a container statement is added as a new member of that object. The metadata object is placed directly inside the container object it modifies and uses a single "@" symbol as the member name. The metadata object representing the attribute list for a leaf statement is added as a sibling name/value pair that is placed directly after the statement it modifies and that has a member name that is the concatenation of the "@" symbol and the statement name.

The metadata object value in both cases is an object containing name/value pairs that describe each of the attributes associated with that container or leaf statement.

```
{
  "container" : {
    "@" : {
      "attribute-name" : attribute-value,
      "attribute-name" : attribute-value
    },
    "statement-name" : "statement-value",
    "@statement-name" : {
      "attribute-name" : attribute-value,
      "attribute-name" : attribute-value
    }
  }
}
```

In the following examples, both the **[edit commit]** hierarchy and the **persist-groups-inheritance** statement have been deactivated. In the CLI, the statements are preceded by the **inactive:** tag. The Junos XML representation includes the **inactive="inactive"** attribute in each of the opening tags for those elements. The JSON mapping stores the attributes in an attribute list. The attribute list for the **[edit commit]** hierarchy is indicated with "@", because it is a container statement. The attribute list for the **persist-groups-inheritance** statement is indicated using "@persist-groups-inheritance", because it is a leaf statement.

CLI Configuration Statements

```
system {
  inactive: commit {
    inactive: persist-groups-inheritance;
  }
}
```

XML Syntax

```
<configuration>
  <system>
    <commit inactive="inactive">
      <persist-groups-inheritance inactive="inactive"/>
    </commit>
  </system>
</configuration>
```

JSON Syntax

```
{
  "configuration" : {
    "system" : {
      "commit" : {
        "@" : {
          "inactive" : true
        },
        "persist-groups-inheritance" : [null],
        "@persist-groups-inheritance" : {
          "inactive" : true
        }
      }
    }
  }
```

```
    }  
  }  
}
```

The attribute list for a specific instance of an object is similar to the attribute list for a container in that it is a name/value pair where the field name is a single "@" symbol, and the value is an object containing name/value pairs that describes each of the attributes. The attribute list is enclosed within the JSON object that identifies that instance and is the first member in the object, followed by the identifier for the object.

```
{  
  "keyword" : [  
    {  
      "@" : {  
        "attribute-name" : attribute-value  
      },  
      "name" : "identifier",  
      ...JSON data for individual characteristics...  
    },  
    /* additional objects */  
  ]  
}
```

In the following example, the ge-0/0/0 interface is protected. In the CLI, the object is preceded by the **protect:** tag. The Junos XML representation includes the **protect="protect"** attribute in the opening tag for that object. The JSON mapping stores the **"protect" : true** attribute in an attribute list that is included in the JSON object identifying that specific interface.

CLI Configuration Statements

```
protect: ge-0/0/0 {  
  unit 0 {  
    family inet {  
      address 198.51.100.1/24;  
    }  
  }  
}
```

XML Syntax

```
<configuration>  
  <interfaces>  
    <interface protect="protect">  
      <name>ge-0/0/0</name>  
      <unit>  
        <name>0</name>  
        <family>  
          <inet>  
            <address>  
              <name>198.51.100.1/24</name>  
            </address>  
          </inet>  
        </family>  
      </unit>  
    </interface>
```

```

    </interfaces>
  </configuration>

```

JSON Syntax

```

{
  "configuration" : {
    "interfaces" : {
      "interface" : [
        {
          "@" : {
            "protect" : true
          },
          "name" : "ge-0/0/0",
          "unit" : [
            {
              "name" : 0,
              "family" : {
                "inet" : {
                  "address" : [
                    {
                      "name" : "198.51.100.1/24"
                    }
                  ]
                }
              }
            }
          ]
        }
      ]
    }
  }
}

```

Mapping for Configuration Comments

A Junos OS configuration can include comments that describe statements in the configuration. Configuration data formatted using ASCII text or Junos XML elements displays comments on the line preceding the statement that the comment modifies. In Junos XML format, the comment string is enclosed in a `<junos:comment>` element.

Comments are indicated using one of two delimiters: the paired strings `/*` and `*/` enclosing the comment, or the `#` character preceding the comment. You can use either delimiter in the comment string when you insert comments in the configuration. If you omit the delimiter, Junos OS automatically inserts `/*` and `*/`.



NOTE: Junos OS preserves any white space included in the comment.

Junos OS configuration data formatted using JSON maps a comment to a name/value pair that is stored as an attribute of the statement that it modifies. The field name is set to `"comment"`, and the value is the comment text string. The comment text string can include either of the two delimiters that indicate a comment. If you omit the delimiter from the comment text string when you load the JSON configuration data, Junos OS automatically adds the `/*` and `*/` delimiters to the comment. You can also create multiline

comments in JSON configuration data by inserting the newline character (`\n`) in the comment string.

The following example shows the formatted ASCII configuration and corresponding JSON syntax for three comments. The example associates one comment with a hierarchy, another comment with an object that has an identifier, and a third comment with a leaf statement.

CLI Configuration Statements

```
protocols {
  /* New backbone area */
  ospf {
    area 0.0.0.0 {
      /* From jnpr1
      to jnpr2 */
      interface so-0/0/0.0 {
        # set by admin
        hello-interval 5;
      }
    }
  }
}
```

JSON Syntax

```
{
  "configuration" : {
    "protocols" : {
      "ospf" : {
        "@" : {
          "comment" : "/* New backbone area */"
        },
        "area" : [
          {
            "name" : "0.0.0.0",
            "interface" : [
              {
                "@" : {
                  "comment" : "/* From jnpr1 \n to jnpr2 */"
                },
                "name" : "so-0/0/0.0",
                "hello-interval" : 5,
                "@hello-interval" : {
                  "comment" : "# set by admin"
                }
              }
            ]
          }
        ]
      }
    }
  }
}
```

Release History Table

Release	Description
16.1R4	Starting in Junos OS Releases 16.1R4, 16.2R2, and 17.1R1, integers in Junos OS configuration data emitted in JSON format are not enclosed in quotation marks. In earlier releases, integers in JSON configuration data were treated as strings and enclosed in quotation marks.
16.1	Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.

PART 2

Managing Junos XML Protocol Sessions

- [Junos XML Protocol Session Overview on page 51](#)
- [Managing Junos XML Protocol Sessions on page 57](#)
- [Junos XML Protocol Tracing Operations on page 95](#)
- [Junos XML Protocol Operations on page 103](#)
- [Junos XML Protocol Processing Instructions on page 123](#)
- [Junos XML Protocol Response Tags on page 125](#)
- [Junos XML Element Attributes on page 139](#)

CHAPTER 4

Junos XML Protocol Session Overview

- [Junos XML Protocol Session Overview on page 51](#)
- [Supported Access Protocols for Junos XML Protocol Sessions on page 52](#)
- [Understanding the Client Application's Role in a Junos XML Protocol Session on page 53](#)
- [Understanding the Request Procedure in a Junos XML Protocol Session on page 54](#)

Junos XML Protocol Session Overview

The Junos XML protocol server communicates with client applications within the context of a Junos XML protocol *session*. The server and client explicitly establish a connection and session before exchanging data and close the session and connection when they are finished.

Each request from the client application and each response from the Junos XML protocol server must constitute a *well-formed* XML document by obeying the structural rules defined in the Junos XML protocol and Junos XML document type definition (DTD) for the kind of information they encode. The client application must produce a well-formed XML document for each request by emitting tag elements in the required order and only in the legal contexts.

Client applications access the Junos XML protocol server using one of the protocols listed in [“Supported Access Protocols for Junos XML Protocol Sessions” on page 52](#). To authenticate with the Junos XML protocol server, a client application uses either a Junos XML protocol-specific mechanism or the access protocol’s standard authentication mechanism, depending on the protocol. After authentication, the Junos XML protocol server uses the Junos OS login usernames and classes configured on the device to determine whether a client application is authorized to make each request.

The following list outlines the basic structure of a Junos XML protocol session:

1. The client application establishes a connection to the Junos XML protocol server and opens the Junos XML protocol session.
2. The Junos XML protocol server and client application exchange initialization information, which is used to determine if they are using compatible versions of the Junos OS and the Junos XML management protocol.

3. The client application sends one or more requests to the Junos XML protocol server and parses its responses.
4. The client application closes the Junos XML protocol session and the connection to the Junos XML protocol server.

Related Documentation

- [Supported Access Protocols for Junos XML Protocol Sessions on page 52](#)
- [Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server on page 57](#)
- [Understanding the Client Application's Role in a Junos XML Protocol Session on page 53](#)

Supported Access Protocols for Junos XML Protocol Sessions

To connect to the Junos XML protocol server, client applications can use the access protocols and associated authentication mechanisms listed in [Table 5 on page 52](#).

Table 5: Supported Access Protocols and Authentication Mechanisms

Access Protocol	Authentication Mechanism
clear-text, a Junos XML protocol-specific access protocol for sending unencrypted text over a Transmission Control Protocol (TCP) connection	Junos XML protocol-specific
SSH	Standard SSH
Outbound SSH	Outbound SSH
Secure Sockets Layer (SSL)	Junos XML protocol-specific
Telnet	Standard Telnet

The SSH and SSL protocols are preferred because they encrypt security information (such as passwords) before transmitting it across the network. Outbound SSH allows you to create an encrypted connection to the device in situations where you cannot connect to the device using standard SSH. The clear-text and Telnet protocols do not encrypt information.

For information about the prerequisites for each access protocol, see [“Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server” on page 57](#). For authentication instructions, see [“Authenticating with the Junos XML Protocol Server for Cleartext or SSL Connections” on page 75](#).

Related Documentation

- [Understanding the Client Application's Role in a Junos XML Protocol Session on page 53](#)
- [Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server on page 57](#)
- [Connecting to the Junos XML Protocol Server on page 68](#)

- [Starting Junos XML Protocol Sessions](#) on page 70
- [Authenticating with the Junos XML Protocol Server for Cleartext or SSL Connections](#) on page 75

Understanding the Client Application's Role in a Junos XML Protocol Session

To create a session and communicate with the Junos XML protocol server, a client application performs the following procedures, which are described in the indicated sections:

1. Satisfies any prerequisites required for a connection, as described in [“Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server”](#) on page 57.
2. Establishes a connection to the Junos XML protocol server on the routing, switching, or security platform, as described in [“Connecting to the Junos XML Protocol Server”](#) on page 68.
3. Starts a Junos XML protocol session, as described in [“Starting Junos XML Protocol Sessions”](#) on page 70.

4. Optionally locks the candidate configuration, creates a private copy of the configuration, or opens an instance of the ephemeral configuration database.

Locking the configuration prevents other users or applications from changing it at the same time. Creating a private copy of the configuration enables the application to make changes without affecting the candidate configuration until the copy is committed. For more information, see [“Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol”](#) on page 86.

For information about the ephemeral configuration database, see [“Understanding the Ephemeral Configuration Database”](#) on page 237 and [“Enabling and Configuring Instances of the Ephemeral Configuration Database”](#) on page 244.

5. Requests operational or configuration information, or changes the configuration, as described in [“Sending Requests to the Junos XML Protocol Server”](#) on page 77.
6. (Optional) Verifies the syntactic correctness of the candidate configuration or private copy before attempting to commit it, as described in [“Verifying Configuration Syntax Using the Junos XML Protocol”](#) on page 217.
7. Commits changes made to the candidate configuration or private copy, as described in [“Committing the Candidate Configuration Using the Junos XML Protocol”](#) on page 218, or commits changes made to an open instance of the ephemeral configuration database, as described in [“Committing an Instance of the Ephemeral Configuration Database Using the NETCONF or Junos XML Protocol”](#) on page 249.
8. Unlocks the candidate configuration if it is locked or closes a private copy of the configuration or an open instance of the ephemeral configuration database.

Other users and applications cannot change the candidate configuration while it remains locked. For more information, see [“Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol”](#) on page 86.

9. Ends the Junos XML protocol session and closes the connection to the device, as described in [“Ending a Junos XML Protocol Session and Closing the Connection”](#) on page 90.

Related Documentation

- [Supported Access Protocols for Junos XML Protocol Sessions](#) on page 52
- [Authenticating with the Junos XML Protocol Server for Cleartext or SSL Connections](#) on page 75
- [Parsing the Junos XML Protocol Server Response](#) on page 80
- [Sample Junos XML Protocol Session](#) on page 91

Understanding the Request Procedure in a Junos XML Protocol Session

You can use the Junos XML management protocol and Junos XML API to request information about the status and the current configuration of a routing, switching, or security platform running Junos OS. The tags for operational requests are defined in the Junos XML API and correspond to Junos OS command-line interface (CLI) operational commands. There is a request tag for many commands in the CLI **show** family of commands.

The tag for configuration requests is the Junos XML protocol **<get-configuration>** tag. It corresponds to the CLI configuration mode **show** command. The Junos XML tag elements that make up the content of both the client application's requests and the Junos XML protocol server's responses correspond to CLI configuration statements, which are described in the Junos OS configuration guides.

In addition to information about the current configuration, client applications can request other configuration-related information, including information about previously committed (rollback) configurations, information about the rescue configuration, or an XML schema representation of the configuration hierarchy.

To request information from the Junos XML protocol server, a client application performs the procedures described in the indicated sections:

1. Establishes a connection to the Junos XML protocol server on the routing, switching, or security platform, as described in [“Connecting to the Junos XML Protocol Server”](#) on page 68.
2. Starts a Junos XML protocol session, as described in [“Starting Junos XML Protocol Sessions”](#) on page 70.
3. Optionally locks the candidate configuration, creates a private copy of the configuration, or opens an instance of the ephemeral configuration database.

Locking the configuration prevents other users or applications from changing it at the same time. Creating a private copy of the configuration enables the application to

make changes without affecting the candidate configuration until the copy is committed. For more information, see [“Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol”](#) on page 86.

For information about the ephemeral configuration database, see [“Understanding the Ephemeral Configuration Database”](#) on page 237 and [“Enabling and Configuring Instances of the Ephemeral Configuration Database”](#) on page 244.

4. Makes any number of requests one at a time, freely intermingling operational and configuration requests. See [“Requesting Operational Information Using the Junos XML Protocol”](#) on page 255 and [“Requesting Configuration Data Using the Junos XML Protocol”](#) on page 270.

The application can also intermix requests with configuration changes.

5. Accepts the tag stream emitted by the Junos XML protocol server in response to each request and extracts its content, as described in [“Parsing the Junos XML Protocol Server Response”](#) on page 80.
6. Unlocks the candidate configuration if it is locked or closes a private copy of the configuration or an open instance of the ephemeral configuration database.

Other users and applications cannot change the candidate configuration while it remains locked. For more information, see [“Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol”](#) on page 86.

7. Ends the Junos XML protocol session and closes the connection to the device, as described in [“Ending a Junos XML Protocol Session and Closing the Connection”](#) on page 90.

Related Documentation

- [Requesting Operational Information Using the Junos XML Protocol](#) on page 255
- [Requesting Configuration Data Using the Junos XML Protocol](#) on page 270
- [Requesting the Complete Configuration Using the Junos XML Protocol](#) on page 302
- [Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol](#) on page 86

CHAPTER 5

Managing Junos XML Protocol Sessions

- [Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server on page 57](#)
- [Connecting to the Junos XML Protocol Server on page 68](#)
- [Starting Junos XML Protocol Sessions on page 70](#)
- [Authenticating with the Junos XML Protocol Server for Cleartext or SSL Connections on page 75](#)
- [Sending Requests to the Junos XML Protocol Server on page 77](#)
- [Parsing the Junos XML Protocol Server Response on page 80](#)
- [Using a Standard API to Parse Response Tag Elements in NETCONF and Junos XML Protocol Sessions on page 82](#)
- [Understanding Character Encoding on Devices Running Junos OS on page 83](#)
- [Handling an Error or Warning in Junos XML Protocol Sessions on page 84](#)
- [Halting a Request in Junos XML Protocol Sessions on page 85](#)
- [Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol on page 86](#)
- [Terminating Junos XML Protocol Sessions on page 89](#)
- [Ending a Junos XML Protocol Session and Closing the Connection on page 90](#)
- [Sample Junos XML Protocol Session on page 91](#)

Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server

To enable a client application to establish a connection to the Junos XML protocol server, you must satisfy the requirements that are applicable to all access protocols as well as your specific access protocol as discussed in the following sections:

- [Prerequisites for All Access Protocols on page 58](#)
- [Prerequisites for Clear-Text Connections on page 59](#)
- [Prerequisites for SSH Connections on page 61](#)
- [Prerequisites for Outbound SSH Connections on page 62](#)

- [Prerequisites for SSL Connections on page 65](#)
- [Prerequisites for Telnet Connections on page 67](#)

Prerequisites for All Access Protocols

A client application must be able to log in to each device on which it establishes a connection with the Junos XML protocol server. The following instructions explain how to create a Junos login account for the application; for detailed information, see the chapter about configuring user access in the *Junos OS Administration Library*. Alternatively, you can skip this section and enable authentication through RADIUS or TACACS+; for instructions, see the chapter about system authentication in the *Junos OS Administration Library*.

To determine whether a login account exists on a device running Junos OS, enter the CLI configuration mode on the device and issue the following commands:

```
[edit system login]
user@host# show user account-name
```

If the appropriate account does not exist, perform the following steps:

1. Include the **user** statement at the **[edit system login]** hierarchy level and specify a username. Also include the **class** statement at the **[edit system login user username]** hierarchy level, and specify a login class that has the permissions required for all actions to be performed by the application. Optionally, include the **full-name** and **uid** statements.

```
[edit system login]
user@host# set user account-name class class-name
```



NOTE: For detailed information about creating user accounts, see the chapter about configuring user access in the *Junos OS Administration Library*.

2. Create a text-based password for the account by including either the **plain-text-password** or **encrypted-password** statement at the **[edit system login user account-name authentication]** hierarchy level.

```
[edit system login]
user@host# edit user account-name authentication
```



NOTE: A text-based password is not strictly necessary if the account is used to access the Junos XML protocol server through SSH with public/private key pairs for authentication, but we recommend that you create one anyway. The key pair alone is sufficient if the account is used only for SSH access, but a password is required if the account is also used for any other type of access (for login on the console, for example). The password is also used—the SSH server prompts for it—if key-based authentication is configured but fails. For information about creating a public/private key pair, see [“Prerequisites for SSH Connections” on page 61](#).

To enter a password as text, issue the following command. You are prompted for the password, which is encrypted before being stored.

```
[edit system login user account-name authentication]
user@host# set plain-text-password
New password: password
Retype new password: password
```

To store a password that you have previously created and hashed using Message Digest 5 (MD5) or Secure Hash Algorithm 1 (SHA-1), issue the following command:

```
[edit system login user account-name authentication]
user@host# set encrypted-password "password"
```

3. Issue the **commit** command.

```
[edit system login user account-name authentication]
user@host# top
[edit]
user@host# commit
```

4. Repeat the preceding steps on each device where the client application establishes Junos XML protocol sessions.
5. Enable the client application to access the password and provide it when the Junos XML protocol server prompts for it. There are several possible methods, including the following:
 - Code the application to prompt the user for a password at startup and to store the password temporarily in a secure manner.
 - Store the password in encrypted form in a secure local-disk location or secured database and code the application to access it.

Prerequisites for Clear-Text Connections

A client application that uses the Junos XML protocol-specific clear-text access protocol sends unencrypted text directly over a TCP connection without using any additional protocol (such as SSH, SSL, or Telnet).



NOTE: Devices running the Junos-FIPS software do not accept Junos XML protocol clear-text connections. We recommend that you do not use the clear-text protocol in a Common Criteria environment. For more information, see the *Secure Configuration Guide for Common Criteria and Junos-FIPS*.

To enable client applications to use the clear-text protocol to connect to the Junos XML protocol server, perform the following steps:

1. Verify that the application can access the TCP software. On most operating systems, TCP is accessible in the standard distribution. Do this on each computer where the application runs.
2. Satisfy the prerequisites discussed in [“Prerequisites for All Access Protocols” on page 58](#).
3. Configure the device running Junos OS to accept clear-text connections from client applications on port 3221 by including the **xnm-clear-text** statement at the **[edit system services]** hierarchy level:

```
[edit]
user@host# set system services xnm-clear-text
```

By default, the Junos XML protocol server supports up to 75 simultaneous clear-text sessions and 150 connection attempts per minute. Optionally, you can include either or both the **connection-limit** statement to limit the number of concurrent sessions and the **rate-limit** statement to limit the number of connection attempts. Both statements accept a value from 1 through 250.

```
[edit]
user@host# set system services xnm-clear-text connection-limit limit
user@host# set system services xnm-clear-text rate-limit limit
```

For more information about the **xnm-clear-text** statement, see the *Junos OS Administration Library*.

4. Commit the configuration:

```
[edit]
user@host# commit
```
5. Repeat Step 2 through Step 4 on each device where the client application establishes Junos XML protocol sessions.

Prerequisites for SSH Connections

To enable a client application to use the SSH protocol to connect to the Junos XML protocol server, perform the following steps:

1. Enable the application to access the SSH software.

If the application uses the Junos XML protocol Perl module provided by Juniper Networks, no action is necessary. As part of the installation procedure for the Perl module, you install a prerequisites package that includes the necessary SSH software. For instructions, see [“Downloading the Junos XML Protocol Perl Client and Prerequisites Package” on page 334](#).

If the application does not use the Junos XML protocol Perl module, obtain the SSH software and install it on the computer where the application runs. For information about obtaining and installing SSH software, see <http://www.ssh.com/> and <http://www.openssh.com/>.

2. Satisfy the prerequisites discussed in [“Prerequisites for All Access Protocols” on page 58](#).

3. (Optional) If you want to use key-based SSH authentication for the application, create a public/private key pair and associate it with the Junos OS login account you created in [“Prerequisites for All Access Protocols” on page 58](#). Perform the following steps:

- a. Working on the computer where the client application runs, issue the **ssh-keygen** command in a standard command shell (not the Junos OS CLI). By providing the appropriate arguments, you encode the public key with either RSA (supported by SSH versions 1 and 2) or the Digital Signature Algorithm (DSA), supported by SSH version 2. For more information, see the man page provided by your SSH vendor for the **ssh-keygen** command. The Junos OS uses SSH version 2 by default but also supports version 1.

% ssh-keygen options

- b. Enable the application to access the public and private keys. One method is to run the **ssh-agent** program on the computer where the application runs.
- c. On the device running Junos OS that needs to accept SSH connections from Junos XML protocol client applications, associate the public key with the Junos login account by including the **load-key-file** statement at the **[edit system login user account-name authentication]** hierarchy level. First, move to that hierarchy level:

```
[edit]
user@host# edit system login user account-name authentication
```

Issue the following command to copy the contents of the specified file onto the device running Junos OS:

```
[edit system login user account-name authentication]
user@host# set load-key-file URL
```

URL is the path to the file that contains one or more public keys. The **ssh-keygen** command by default stores each public key in a file in the **.ssh** subdirectory of the

user home directory; the filename depends on the encoding (DSA or RSA) and SSH version. For information about specifying URLs, see the *CLI User Guide*.

Alternatively, you can include one or both of the **ssh-dsa** and **ssh-rsa** statements at the **[edit system login user account-name authentication]** hierarchy level. We recommend using the **load-key-file** statement, however, because it eliminates the need to type or cut and paste the public key on the command line. For more information about the **ssh-dsa** and **ssh-rsa** statements, see the *Junos OS Administration Library*.

4. Configure the device running Junos OS to accept SSH connections by including the **ssh** statement at the **[edit system services]** hierarchy level. This statement enables SSH access for all users and applications, not just Junos XML protocol client applications.

```
[edit system login user account-name authentication]
user@host# top
[edit]
user@host# set system services ssh
```

5. Commit the configuration:

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

6. Repeat Step 1 on each computer where the application runs, and Step 2 through Step 5 on each device to which the application connects.

Prerequisites for Outbound SSH Connections

The outbound SSH feature allows the initiation of an SSH session between devices running Junos OS and Network and System Management servers where client-initiated TCP/IP connections are blocked (for example, when the device is behind a firewall). To configure outbound SSH, you add an **outbound-ssh** configuration statement to the device. Once configured and committed, the device running Junos OS will begin to initiate outbound SSH sessions with the configured management clients. Once the outbound SSH session is initialized and the connection is established, the management server initiates the SSH sequence as the client and the device running Junos OS, acting as the server, authenticates the client.

Setting up outbound SSH involves:

- Configuring the device running Junos OS for outbound SSH
- Configuring the management server for outbound SSH.

To configure the device for outbound SSH:

1. Satisfy the prerequisites discussed in [“Prerequisites for All Access Protocols” on page 58](#).
2. In the **[edit system services ssh]** hierarchy level, set the SSH protocol to v2:

```
[edit system services ssh]
```

```
set protocol-version v2
```

3. Generate/obtain a public/private key pair for the device running Junos OS. This key pair will be used to encrypt the data transferred across the SSH connection. For more information on generating key pairs, see the *Junos OS Administration Library*.
4. If the public key will be installed on the application management system manually, transfer the public key to the NSM server.
5. Add the following **outbound-ssh** statement at the **[edit system services]** hierarchy level:

```
[edit system services]
outbound-ssh {
  client client-id {
    address {
      port port-number;
      retry number;
      timeout seconds;
    }
    device-id device-id;
    keep-alive {
      retry number;
      timeout seconds;
    }
    reconnect-strategy (in-order | sticky);
    secret password;
    services netconf;
  }
}
```

The options are as follows:

- **address**—(Required) Hostname or IPv4 or IPv6 address of the management server. You can list multiple clients by adding each client's IP address or hostname along with the following connection parameters.
 - **port *port-number***—Outbound SSH port for the client. The default is port 22.
 - **retry *number***— Number of times the device attempts to establish an outbound SSH connection. The default is three tries.
 - **timeout *seconds***—Amount of time, in seconds, that the device running Junos OS attempts to establish an outbound SSH connection. The default is 15 seconds.
- **client *client-id***—(Required) Identifies the **outbound-ssh** configuration stanza on the device. Each **outbound-ssh** stanza represents a single outbound SSH connection. This attribute is not sent to the client.
- **device-id *device-id***—(Required) Identifies the device running Junos OS to the client during the initiation sequence.

- **keep-alive**—(Optional) Specify that the device send keepalive messages to the management server. To configure the keepalive message, you must set both the **timeout** and **retry** attributes.
 - **retry number**—Number of keepalive messages the device sends without receiving a response from the management server before the current SSH connection is terminated. The default is three tries.
 - **timeout seconds**—Amount of time, in seconds, that the server waits for data before sending a keepalive signal. The default is 15 seconds.
- **reconnect-strategy (in-order | sticky)**—(Optional) Specify the method the router or switch uses to reestablish a disconnected outbound SSH connection. Two methods are available:
 - **in-order**—Specify that the router or switch first attempt to establish an outbound SSH session based on the management server address list. The router or switch attempts to establish a session with the first server on the list. If this connection is not available, the router or switch attempts to establish a session with the next server, and so on down the list until a connection is established.
 - **sticky**—Specify that the router or switch first attempt to reconnect to the management server that it was last connected to. If the connection is unavailable, it attempts to establish a connection with the next client on the list and so forth until a connection is made.

When reconnecting to a client, the device running Junos OS attempts to reconnect to the client based on the **retry** and **timeout** values for each of the clients listed in the configuration management server list.

- **secret password**—(Optional) Public SSH host key of the device running Junos OS. If added to the **outbound-ssh** statement, during the initialization of the outbound SSH service, the router or switch passes its public key to the management server. This is the recommended method of maintaining a current copy of the router's or switch's public key.
- **services**—(Required) Specifies the services available for the session. Currently, NETCONF is the only service available.

6. Commit the configuration:

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

To set up the configuration management server:

1. Satisfy the prerequisites discussed in [“Prerequisites for All Access Protocols” on page 58](#).
2. Enable the application to access the SSH software.
 - If the application uses the Junos XML protocol Perl module provided by Juniper Networks, no action is necessary. As part of the installation procedure for the Perl module, you install a prerequisites package that includes the necessary

SSH software. For instructions, see [“Downloading the Junos XML Protocol Perl Client and Prerequisites Package” on page 334](#).

- If the application does not use the Junos XML protocol Perl module, obtain the SSH software and install it on the computer where the application runs. For information about obtaining and installing SSH software, see <http://www.ssh.com/> and <http://www.openssh.com/>.
3. (Optional) Manually install the device's public key for use with the SSH connection.
 4. Configure the client system to receive and process initialization broadcast requests. The initialization requests use the following syntax:
 - If the secret attribute is configured, the device running Junos OS will send its public SSH key along with the initialization sequence (recommended method). When the key has been received, the client needs to determine what to do with the device's public key. We recommend that you replace any current public SSH key for the device with the new key. This ensures that the client always has the current key available for authentication.

```
MSG-ID: DEVICE-CONN-INFO\r\n
MSG-VER: V1\r\n
DEVICE-ID: <device-id>\r\n
HOST-KEY: <pub-host-key>\r\n
HMAC: <HMAC(pub-SSH-host-key, <secret>)>\r\n
```

- If the secret attribute is not configured, the device does not send its public SSH key along with the initialization sequence. You need to manually install the current public SSH key for the device.

```
MSG-ID: DEVICE-CONN-INFO\r\n
MSG-VER: V1\r\n
DEVICE-ID: <device-id>\r\n
```

Prerequisites for SSL Connections

To enable a client application to use the SSL protocol to connect to the Junos XML protocol server, perform the following steps:

1. Enable the application to access the SSL software.

If the application uses the Junos XML protocol Perl module provided by Juniper Networks, no action is necessary. As part of the installation procedure for the Perl module, you install a prerequisites package that includes the necessary SSL software. For instructions, see [“Downloading the Junos XML Protocol Perl Client and Prerequisites Package” on page 334](#).

If the application does not use the Junos XML protocol Perl module, obtain the SSL software and install it on the computer where the application runs. For information about obtaining and installing the SSL software, see <http://www.openssl.org/>.

2. Satisfy the prerequisites discussed in [“Prerequisites for All Access Protocols” on page 58](#).

3. Use one of the following two methods to obtain an authentication certificate in privacy-enhanced mail (PEM) format:
 - Request a certificate from a certificate authority; these agencies usually charge a fee.
 - Working on the computer where the client application runs, issue the following **openssl** command in a standard command shell (not the Junos OS CLI). The command generates a self-signed certificate and an unencrypted 1024-bit RSA private key, and writes them to the file called **certificate-file.pem** in the working directory. The command appears here on two lines only for legibility:

```
% openssl req -x509 -nodes -newkey rsa:1024 \
    -keyout certificate-file.pem -out certificate-file.pem
```

4. Import the certificate onto the device running Junos OS by including the **local** statement at the **[edit security certificates]** hierarchy level and the **load-key-file** statement at the **[edit security certificates local certificate-name]** hierarchy level.

```
[edit]
user@host# edit security certificates local certificate-name
```

```
[edit security certificates local certificate-name]
user@host# set load-key-file URL-or-path
```

certificate-name is a name you choose to identify the certificate uniquely (for example, **junos-xml-protocol-ssl-client-hostname**, where **hostname** is the computer where the client application runs).

URL-or-path specifies the file that contains the paired certificate and private key (if you issued the **openssl** command in Step 3, the **certificate-name.pem** file). Specify either the URL to its location on the client computer or a pathname on the local disk (if you have already used another method to copy the certificate file to the device's local disk). For more information about specifying URLs and pathnames, see the *CLI User Guide*.



NOTE: The CLI expects the private key in the **URL-or-path** file to be unencrypted. If the key is encrypted, the CLI prompts you for the passphrase associated with it, decrypts it, and stores the unencrypted version.

The **set-load-key-file URL-or-path** command copies the contents of the certificate file into the configuration. When you view the configuration, the CLI displays the string of characters that constitute the private key and certificate, marking them as **SECRET-DATA**. The **load-key-file** keyword is not recorded in the configuration.

5. Configure the device running Junos OS to accept SSL connections from Junos XML protocol client applications on port 3220 by including the **xnm-ssl** statement at the **[edit system services]** hierarchy level.

```
[edit security certificates local certificate-name]
```

```

user@host# top
[edit]
user@host# set system services xnm-ssl local-certificate certificate-name

```

certificate-name is the unique name you assigned to the certificate in Step 4.

By default, the Junos XML protocol server supports up to 75 simultaneous SSL sessions and 150 connection attempts per minute. Optionally, you can include either or both the **connection-limit** statement to limit the number of concurrent sessions and the **rate-limit** statement to limit connection attempts. Both statements accept a value from 1 through 250.

```

[edit]
user@host# set system services xnm-ssl connection-limit limit
user@host# set system services xnm-ssl rate-limit limit

```

For more information about the **xnm-ssl** statement, see the *Junos OS Administration Library*.

6. Commit the configuration:

```

[edit]
user@host# commit

```

7. Repeat Step 1 on each computer where the client application runs, and Step 2 through Step 6 on each device to which the client application connects.

Prerequisites for Telnet Connections

To enable a client application to use the Telnet protocol to access the Junos XML protocol server, perform the steps described in this section.

Devices running the Junos-FIPS software do not accept Telnet connections. We recommend that you do not use the Telnet protocol in a Common Criteria environment. For more information, see the *Secure Configuration Guide for Common Criteria and Junos-FIPS*.

1. Verify that the application can access the Telnet software. On most operating systems, Telnet is accessible in the standard distribution.
2. Satisfy the prerequisites discussed in "[Prerequisites for All Access Protocols](#)" on [page 58](#).
3. Configure the device running Junos OS to accept Telnet connections by including the **telnet** statement at the **[edit system services]** hierarchy level. This statement enables Telnet access for all users and applications, not just Junos XML protocol client applications.

```

[edit]

```

```
user@host# set system services telnet
```

4. Repeat Step 1 on each computer where the application runs, and Step 2 and Step 3 on each device to which the application connects.

Related Documentation

- [Understanding the Client Application's Role in a Junos XML Protocol Session on page 53](#)
- [Supported Access Protocols for Junos XML Protocol Sessions on page 52](#)
- [Connecting to the Junos XML Protocol Server on page 68](#)
- [Starting Junos XML Protocol Sessions on page 70](#)
- [Authenticating with the Junos XML Protocol Server for Cleartext or SSL Connections on page 75](#)

Connecting to the Junos XML Protocol Server

You can connect to the Junos XML protocol server through the Junos OS command-line interface (CLI) or through a client application. The following sections provide details for each method:

- [Connecting to the Junos XML Protocol Server from the CLI on page 68](#)
- [Connecting to the Junos XML Protocol Server from the Client Application on page 69](#)

Connecting to the Junos XML Protocol Server from the CLI

The Junos XML management protocol and Junos XML API are primarily intended for use by client applications; however, for testing purposes you can establish an interactive Junos XML protocol session and type commands in a shell window. To connect to the Junos XML protocol server from the CLI operational mode, issue the **junoscript interactive** command (the **interactive** option causes the Junos XML protocol server to echo what you type):

```
user@host> junoscript interactive
```

To begin a Junos XML protocol session over the connection, emit the initialization PI and tag that are described in “[Starting Junos XML Protocol Sessions](#)” on page 70. You can then type sequences of tag elements that represent operational and configuration operations. To eliminate typing errors, save complete tag element sequences in a file and use a cut-and-paste utility to copy the sequences to the shell window.



NOTE: When you close the connection to the Junos XML protocol server (for example, by emitting the `<request-end-session/>` and `</junoscript>` tags), the routing platform completely closes your connection instead of returning you to the CLI operational mode prompt. For more information about ending a Junos XML protocol session, see “[Ending a Junos XML Protocol Session and Closing the Connection](#)” on page 90.

Connecting to the Junos XML Protocol Server from the Client Application

For a client application to connect to the Junos XML protocol server and open a session, you must first satisfy the prerequisites described in [“Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server” on page 57](#).

When the prerequisites are satisfied, an application written in Perl can most efficiently establish a connection and open a session by using the Junos XML protocol Perl module provided by Juniper Networks. For more information, see [“Understanding the Junos XML Protocol Perl Distribution and Sample Scripts” on page 333](#).

A client application that does not use the Junos XML protocol Perl module connects to the Junos XML protocol server by opening a socket or other communications channel to the Junos XML protocol server device, invoking one of the remote-connection routines appropriate for the programming language and access protocol that the application uses.

What the client application does next depends on which access protocol it is using:

- If using the clear-text or SSL protocol, the client application performs the following steps:
 1. Emits the initialization PI and tag, as described in [“Starting Junos XML Protocol Sessions” on page 70](#).
 2. Authenticates with the Junos XML protocol server, as described in [“Authenticating with the Junos XML Protocol Server for Cleartext or SSL Connections” on page 75](#).
- If using the SSH or Telnet protocol, the client application performs the following steps:
 1. Uses the protocol’s built-in authentication mechanism to authenticate.
 2. Issues the **junoscript** command to request that the Junos XML protocol server convert the connection into a Junos XML protocol session. For a C programming language example, see [“Establishing a Junos XML Protocol Session Using C Client Applications” on page 369](#) and [“Accessing and Editing Device Configurations Using Junos XML Protocol C Client Applications” on page 370](#).
 3. Emits the initialization PI and tag, as described in [“Starting Junos XML Protocol Sessions” on page 70](#).

Related Documentation

- [Understanding the Client Application’s Role in a Junos XML Protocol Session on page 53](#)
- [Supported Access Protocols for Junos XML Protocol Sessions on page 52](#)
- [Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server on page 57](#)
- [Starting Junos XML Protocol Sessions on page 70](#)

- [Terminating Junos XML Protocol Sessions on page 89](#)
- [Ending a Junos XML Protocol Session and Closing the Connection on page 90](#)
- [Sample Junos XML Protocol Session on page 91](#)

Starting Junos XML Protocol Sessions

Each Junos XML protocol session begins with a handshake in which the Junos XML protocol server and the client application specify the version of XML and the version of the Junos XML management protocol they are using. Each party parses the version information emitted by the other, using it to determine whether they can communicate successfully. Specifically, the client application emits an `<?xml?>` PI and an opening `<junoscript>` tag. The following sections describe how to start a Junos XML protocol session:

- [Emitting the `<?xml?>` PI on page 70](#)
- [Emitting the Opening `<junoscript>` Tag on page 71](#)
- [Parsing the Junos XML Protocol Server's `<?xml?>` PI on page 72](#)
- [Parsing the Junos XML Protocol Server's Opening `<junoscript>` Tag on page 73](#)
- [Verifying Software Compatibility on page 74](#)

Emitting the `<?xml?>` PI

The client application begins by emitting an `<?xml?>` PI.



.....

NOTE: In the following example (and in all examples in this document of tag elements emitted by a client application), bold font is used to highlight the part of the tag sequence that is discussed in the text.

.....

```
<?xml version="version" encoding="encoding"?>
```

The attributes are as follows. For a list of the attribute values that are acceptable in the current version of the Junos XML management protocol, see [“Verifying Software Compatibility” on page 74](#).

- **version**—The version of XML with which tag elements emitted by the client application comply
- **encoding**—The standardized character set that the client application uses and can understand

In the following example of a client application's `<?xml?>` PI, the `version="1.0"` attribute indicates that the application is emitting tag elements that comply with the XML 1.0 specification. The `encoding="us-ascii"` attribute indicates that the client application is using the 7-bit ASCII character set standardized by the American National Standards Institute (ANSI). For more information about ANSI standards, see <http://www.ansi.org/>.

```
<?xml version="1.0" encoding="us-ascii"?>
```



NOTE: If the application fails to emit the `<?xml?>` PI before emitting the opening `<junoscript>` tag, the Junos XML protocol server emits an error message and immediately closes the session and connection.

Emitting the Opening `<junoscript>` Tag

The client application then emits its opening `<junoscript>` tag, which has the following syntax (and appears here on two lines only for legibility):

```
<junoscript version="version" hostname="hostname" junos:key="key"
      release="release-code">
```

The attributes are as follows. For a list of the attribute values that are acceptable in the current version of the Junos XML management protocol, see [“Verifying Software Compatibility” on page 74](#).

version—(Required) Specifies the version of the Junos XML management protocol that the client application is using.

hostname—(Optional) Names the machine on which the client application is running. The information is used only when diagnosing problems. The Junos XML protocol does not include support for establishing trusted-host relationships or otherwise altering Junos XML protocol server behavior depending on the client hostname.

junos:key—(Optional) Requests that the Junos XML protocol server indicate whether a child configuration element is an identifier for its parent element. The only acceptable value is "key". For more information, see [“Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol” on page 295](#).

release—(Optional) Identifies the Junos OS Release (and by implication, the Junos XML API) for which the client application is designed. The value of this attribute indicates that the client application can interoperate successfully with a Junos XML protocol server that also supports that version of the Junos XML API. In other words, it indicates that the client application emits request tag elements from that API and knows how to parse response tag elements from it. If the application does not include this attribute, the Junos XML protocol server emits tag elements from the Junos XML API that it supports.

For the value of the **release** attribute, use the standard notation for Junos OS version numbers. For example, the value 18.1R1 represents the initial version of Junos OS Release 18.1.

In the following example of a client application's opening `<junoscript>` tag, the `version="1.0"` attribute indicates that it is using Junos XML protocol version 1.0. The `hostname="client1"` attribute indicates that the client application is running on the machine called client1. The `release="18.1R1"` attribute indicates that the switch, router, or security device is running the initial version of Junos OS Release 18.1.

```
<junoscript version="1.0" hostname="client1" release="18.1R1">
```



NOTE: If the application fails to emit the `<?xml?>` PI before emitting the opening `<junoscript>` tag, the Junos XML protocol server emits an error message similar to the following and immediately closes the session and connection:

```
<rpc-reply>
  <xnm:error xmlns="http://xml.juniper.net/xnm/1.1/xnm" \
    xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
    <message>
      communication error while exchanging credentials
    </message>
  </xnm:error>
</rpc-reply>
<!-- session end at YYYY-MM-DD hh:mm:ss TZ -->
</junoscript>
```

For more information about the `<xnm:error>` tag, see [“Handling an Error or Warning in Junos XML Protocol Sessions” on page 84](#).

Parsing the Junos XML Protocol Server's `<?xml?>` PI

When the Junos XML protocol session begins, the Junos XML protocol server emits an `<?xml?>` PI and an opening `<junoscript>` tag.

The syntax for the `<?xml?>` PI is as follows:

```
<?xml version="version" encoding="encoding"?>
```

The attributes are as follows. For a list of the attribute values that are acceptable in the current version of the Junos XML management protocol, see [“Verifying Software Compatibility” on page 74](#).

version—The version of XML with which tag elements emitted by the Junos XML protocol server comply

encoding—The standardized character set that the Junos XML protocol server uses and can understand

In the following example of a Junos XML protocol server's `<?xml?>` PI, the `version="1.0"` attribute indicates that the server is emitting tag elements that comply with the XML 1.0 specification. The `encoding="us-ascii"` attribute indicates that the server is using the 7-bit ASCII character set standardized by ANSI. For more information about ANSI standards, see <http://www.ansi.org/>.

```
<?xml version="1.0" encoding="us-ascii"?>
```


Parsing the Junos XML Protocol Server's Opening <junoscript> Tag

After emitting the <?xml?> PI, the server then emits its opening <junoscript> tag, which has the following form (the tag appears on multiple lines only for legibility):

```
<junoscript xmlns="namespace-URL" xmlns:junos="namespace-URL" \
  schemaLocation="namespace-URL" os="JUNOS" \
  release="release-code" hostname="hostname" version="version">
```

The attributes are as follows:

hostname—The name of the device on which the Junos XML protocol server is running.

os—The operating system of the device on which the Junos XML protocol server is running. The value is always JUNOS.

release—The identifier for the version of the Junos OS from which the Junos XML protocol server is derived and that it is designed to understand. It is presumably in use on the device where the Junos XML protocol server is running. The value of the **release** attribute uses the standard notation for Juniper Networks software version numbers. For example, the value 18.1R1 represents the initial version of Junos OS Release 18.1.

schemaLocation—The XML namespace for the XML Schema-language representation of the Junos OS configuration hierarchy.

version—The version of the Junos XML management protocol that the Junos XML protocol server is using.

xmlns—The XML namespace for the tag elements enclosed by the <junoscript> tag element that do not have a prefix on their names (that is, the default namespace for Junos XML tag elements). The value is a URL of the form `http://xml.juniper.net/xnm/version/xnm`, where *version* is a string such as 1.1.

xmlns:junos—The XML namespace for the tag elements enclosed by the <junoscript> tag element that have the **junos:** prefix on their names. The value is a URL of the form `http://xml.juniper.net/junos/release-code/junos`, where *release-code* is the standard string that represents a release of the Junos OS. For example, the value 18.1R1 represents the initial version of Junos OS Release 18.1.

In the following example of a Junos XML protocol server's opening <junoscript> tag, the **version** attribute indicates that the server is using Junos XML protocol version 1.0, and the **hostname** attribute indicates that the router's name is big-device. The **os** and **release** attributes indicate that the device is running the initial version of Junos OS Release 18.1. The **xmlns** attribute indicate that the default namespace for Junos XML tag elements is `http://xml.juniper.net/xnm/1.1/xnm`. The **xmlns:junos** attribute indicates that the namespace for tag elements that have the **junos:** prefix is `http://xml.juniper.net/junos/18.1R1/junos`. The tag appears on multiple lines only for legibility.

```
<junoscript xmlns="http://xml.juniper.net/xnm/1.1/xnm" \
  xmlns:junos="http://xml.juniper.net/junos/18.1R1/junos" \
  schemaLocation="http://xml.juniper.net/junos/18.1R1/junos" os="JUNOS" \
```

```
release="18.1R1.8" hostname="big-device" version="1.0">
```

Verifying Software Compatibility

Exchanging the `<?xml?>` and `<junoscript>` elements enables a client application and the Junos XML protocol server to determine if they are running different versions of the software used during a Junos XML protocol session. Different versions are sometimes incompatible, and by Junos XML protocol convention the party running the later version of software determines how to handle any incompatibility. For fully automated performance, include code in the client application that determines if its version of software is later than that of the Junos XML protocol server. Decide which of the following options is appropriate when the application's version is more recent, and implement the corresponding response:

- Ignore differences in Junos version, and do not alter the client application's behavior to accommodate the Junos XML protocol server. A difference in Junos versions does not necessarily make the server and client incompatible, so this is often a valid approach.
- Alter standard behavior to be compatible with the Junos XML protocol server. If the client application is running a later version of the Junos OS, for example, it can choose to emit only tag elements that represent the software features available in the Junos XML protocol server's version of the Junos OS.
- End the Junos XML protocol session and terminate the connection. This is appropriate if you decide that it is not practical to accommodate the Junos XML protocol server's version of software. For instructions, see [“Ending a Junos XML Protocol Session and Closing the Connection” on page 90](#).

[Table 6 on page 74](#) specifies the PI or opening tag and attribute used to convey version information during Junos XML protocol session initialization in version 1.0 of the Junos XML management protocol.

Table 6: Junos XML Protocol version 1.0 PI and Opening Tag

Software and Versions	PI or Tag	Attribute
XML 1.0	<code><?xml?></code>	<code>version="1.0"</code>
ANSI-standardized 7-bit ASCII character set	<code><?xml?></code>	<code>encoding="us-ascii"</code>
Junos XML protocol 1.0	<code><junoscript></code>	<code>version="1.0"</code>
Junos OS Release	<code><junoscript></code>	<code>release="m.nZb"</code> For example: <code>release="10.3R1"</code>

Related Documentation

- [Understanding the Client Application's Role in a Junos XML Protocol Session on page 53](#)
- [Sending Requests to the Junos XML Protocol Server on page 77](#)
- [Ending a Junos XML Protocol Session and Closing the Connection on page 90](#)

- [Sample Junos XML Protocol Session on page 91](#)

Authenticating with the Junos XML Protocol Server for Cleartext or SSL Connections

A client application that uses cleartext or SSL protocol must authenticate with the Junos XML protocol server. (Applications that use the SSH or Telnet protocol use the protocol's built-in authentication mechanism before emitting initialization tag elements, as described in [“Connecting to the Junos XML Protocol Server” on page 68.](#))

See the following sections:

- [Submitting an Authentication Request on page 75](#)
- [Interpreting the Authentication Response on page 76](#)

Submitting an Authentication Request

The client application begins the authentication process by emitting an `<rpc>` tag enclosing the `<request-login>` element. In the `<request-login>` element, it encloses the `<username>` element to specify the Junos OS account (username) under which to establish the connection. The account must already be configured on the Junos XML protocol server device, as described in [“Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server” on page 57.](#) You can choose whether or not the application provides the account password as part of the initial tag sequence.



NOTE: Starting in Junos OS Releases 13.3R7, 14.1R6, 14.2R4, 15.1R2, and 16.1R1, any XML special characters in the username or password elements of a `<request-login>` RPC request must be escaped. The following five symbols are considered special characters: greater than (`>`), less than (`<`), single quote (`'`), double quote (`"`), and ampersand (`&`). Both entity references and character references are acceptable escape sequence formats. For example, `&` and `&` are valid representations of an ampersand.

Providing the Password with the Username

To provide the password along with the username, the application emits the following tag sequence:

```
<rpc>
  <request-login>
    <username>username</username>
    <challenge-response>password</challenge-response>
  </request-login>
</rpc>
```

This tag sequence is appropriate if the application automates access to routing, switching, or security platform information and does not interact with users, or obtains the password from a user before beginning the authentication process.

Providing Only the Username To omit the password and specify only the username, the application emits the following tag sequence:

```
<rpc>
  <request-login>
    <username>username</username>
  </request-login>
</rpc>
```

This tag sequence is appropriate if the application does not obtain the password until the authentication process has already begun. In this case, the Junos XML protocol server returns the **<challenge>** tag within an **<rpc-reply>** element to request the password associated with the username. The element encloses the **Password:** string, which the client application can forward to the screen as a prompt for a user. The **echo="no"** attribute in the opening **<challenge>** tag specifies that the password string typed by the user does not echo on the screen. The tag sequence is as follows:

```
<rpc-reply xmlns:junos="URL">
  <challenge echo="no">Password:</challenge>
</rpc-reply>
```

The client application obtains the password and emits the following tag sequence to forward it to the Junos XML protocol server:

```
<rpc>
  <request-login>
    <username>username</username>
    <challenge-response>password</challenge-response>
  </request-login>
</rpc>
```

Interpreting the Authentication Response

After it receives the username and password, the Junos XML protocol server emits the **<authentication-response>** element to indicate whether the authentication attempt is successful.

Server Response When Authentication Succeeds If the password is correct, the authentication attempt succeeds and the Junos XML protocol server emits the following tag sequence:

```
<rpc-reply xmlns:junos="URL">
  <authentication-response>
    <status>success</status>
    <message>username</message>
    <login-name>remote-username</login-name>
  </authentication-response>
</rpc-reply>
```

The **<message>** element contains the Junos username under which the connection is established.

The **<login-name>** element contains the username that the client application provided to an authentication utility such as RADIUS or TACACS+. This element appears only if the username differs from the username contained in the **<message>** element.

The Junos XML protocol session begins, as described in [“Starting Junos XML Protocol Sessions” on page 70](#).

Server Response When Authentication Fails

If the password is not correct or the `<request-login>` element is otherwise malformed, the authentication attempt fails and the Junos XML protocol server emits the following tag sequence:

```
<rpc-reply xmlns:junos="URL">
  <authentication-response>
    <status>fail</status>
    <message>error-message</message>
  </authentication-response>
</rpc-reply>
```

The `error-message` string in the `<message>` element explains why the authentication attempt failed. The Junos XML protocol server emits the `<challenge>` tag up to two more times before rejecting the authentication attempt and closing the connection.

Release History Table

Release	Description
13.3R7	Starting in Junos OS Releases 13.3R7, 14.1R6, 14.2R4, 15.1R2, and 16.1R1, any XML special characters in the username or password elements of a <code><request-login></code> RPC request must be escaped.

Related Documentation

- [Understanding the Client Application's Role in a Junos XML Protocol Session on page 53](#)
- [Supported Access Protocols for Junos XML Protocol Sessions on page 52](#)
- [Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server on page 57](#)
- [Connecting to the Junos XML Protocol Server on page 68](#)
- [<request-login> on page 120](#)

Sending Requests to the Junos XML Protocol Server

In a Junos XML protocol session with a device running Junos OS, a client application initiates a request by emitting the opening `<rpc>` tag, one or more tag elements that represent the particular request, and the closing `</rpc>` tag, in that order.

```
<rpc>
  <!--tag elements representing a request-->
</rpc>
```

The application encloses each request in its own separate pair of opening `<rpc>` and closing `</rpc>` tags. Each request must constitute a well-formed XML document by including only compliant and correctly ordered tag elements. The Junos XML protocol server ignores any newline characters, spaces, or other white space characters that occur between tag elements in the tag stream, but it preserves white space within tag elements.

Optionally, a client application can include one or more attributes of the form ***attribute-name="value"*** in the opening **<rpc>** tag for each request. The Junos XML protocol server echoes each attribute, unchanged, in the opening **<rpc-reply>** tag in which it encloses its response.

A client application can use this feature to associate requests and responses by including an attribute in each opening **<rpc>** request tag that assigns a unique identifier. The Junos XML protocol server echoes the attribute in its opening **<rpc-reply>** tag, making it easy to map the response to the initiating request. The client application can freely define attribute names, except as described in the following note.



NOTE: The `xmlns:junos` attribute name is reserved. The Junos XML protocol server sets the attribute to an appropriate value on the opening **<rpc-reply>** tag, so client applications must not emit it in the opening **<rpc>** tag.

Although operational and configuration requests conceptually belong to separate classes, a Junos XML protocol session does not have distinct modes that correspond to CLI operational and configuration modes. Each request tag is enclosed within its own **<rpc>** tag, so a client application can freely alternate operational and configuration requests. A client application can make three classes of requests:

- [Operational Requests on page 78](#)
- [Configuration Information Requests on page 79](#)
- [Configuration Change Requests on page 79](#)

Operational Requests

Operational requests are requests for information about the status of a device running Junos OS. Operational requests correspond to the Junos OS CLI operational mode commands. The Junos XML API defines a request tag for many CLI commands. For example, the **<get-interface-information>** tag corresponds to the **show interfaces** command, and the **<get-chassis-inventory>** tag requests the same information as the **show chassis hardware** command.

The following RPC requests detailed information about interface ge-2/3/0:

```
<rpc>
  <get-interface-information>
    <interface-name>ge-2/3/0</interface-name>
    <detail/>
  </get-interface-information>
</rpc>
```

For more information about operational requests, see [“Requesting Operational Information Using the Junos XML Protocol” on page 255](#). For information about the Junos XML request tag elements available in the current Junos OS Release, see the *Junos XML API Operational Developer Reference* and the [XML API Explorer](#).

Configuration Information Requests

Configuration information requests are requests for information about the device's candidate configuration, a private configuration, the ephemeral configuration, or the committed configuration (the one currently in active use on the routing, switching, or security platform). The candidate and committed configurations diverge when there are uncommitted changes to the candidate configuration.

The Junos XML protocol defines the **<get-configuration>** operation for retrieving configuration information. The Junos XML API defines a tag element for every container and leaf statement in the configuration hierarchy.

The following example shows how to request information about the **[edit system login]** hierarchy level in the candidate configuration:

```
<rpc>
  <get-configuration>
    <configuration>
      <system>
        <login/>
      </system>
    </configuration>
  </get-configuration>
</rpc>
```

For more information about configuration information requests, see [“Requesting Configuration Data Using the Junos XML Protocol” on page 270](#). For a summary of Junos XML configuration tag elements, see the *Junos XML API Configuration Developer Reference* and the [XML API Explorer](#).

Configuration Change Requests

Configuration change requests are requests to change the configuration, or to commit those changes to put them into active use on the device running Junos OS. The Junos XML protocol defines the **<load-configuration>** operation for changing configuration information. The Junos XML API defines a tag element for every CLI configuration statement described in the Junos OS configuration guides.

The following example shows how to create a new Junos OS user account called **admin** at the **[edit system login]** hierarchy level in the candidate configuration:

```
<rpc>
  <load-configuration>
    <configuration>
      <system>
        <login>
          <user>
            <name>admin</name>
            <full-name>Administrator</full-name>
            <class>superuser</class>
          </user>
        </login>
      </system>
    </configuration>
```

```
</load-configuration>
</rpc>
```

For more information about configuration change requests, see [“Requesting Configuration Changes Using the Junos XML Protocol” on page 160](#) and [“Committing the Candidate Configuration Using the Junos XML Protocol” on page 218](#). For a summary of Junos XML configuration tag elements, see the *Junos XML API Configuration Developer Reference* and the [XML API Explorer](#).

Related Documentation

- [Understanding the Client Application’s Role in a Junos XML Protocol Session on page 53](#)
- [XML and Junos XML Management Protocol Conventions Overview on page 11](#)
- [Parsing the Junos XML Protocol Server Response on page 80](#)
- [Handling an Error or Warning in Junos XML Protocol Sessions on page 84](#)
- [Halting a Request in Junos XML Protocol Sessions on page 85](#)

Parsing the Junos XML Protocol Server Response

In a Junos XML protocol session with a device running Junos OS, a client application sends RPCs to the Junos XML protocol server to request information from and manage the configuration on the device. The Junos XML protocol server encloses its response to each client request in a separate pair of opening **<rpc-reply>** and closing **</rpc-reply>** tags. Each response constitutes a well-formed XML document.

```
<rpc-reply xmlns:junos=""http://xml.juniper.net/junos/release/junos"">
  <!-- tag elements representing a response -->
</rpc-reply>
```

The **xmlns:junos** attribute in the opening **<rpc-reply>** tag defines the default namespace for the enclosed Junos XML tag elements that are qualified by the **junos:** prefix. The *release* variable in the URI represents the Junos OS release that is running on the Junos XML protocol server device, for example 18.1R1

The **<rpc-reply>** tag element occurs only within the **<junoscript>** element. Client applications must include code for parsing the stream of response tag elements coming from the Junos XML protocol server, either processing them as they arrive or storing them until the response is complete. The Junos XML protocol server returns three classes of responses:

- [Operational Responses on page 80](#)
- [Configuration Information Responses on page 81](#)
- [Configuration Change Responses on page 82](#)

Operational Responses

Operational responses are responses to requests for information about the status of a switching, routing, or security platform. They correspond to the output from CLI operational commands.

The Junos XML API defines response tag elements for all defined operational request tag elements. For example, the Junos XML protocol server returns the information requested by the `<get-interface-information>` tag in a response tag called `<interface-information>`, and returns the information requested by the `<get-chassis-inventory>` tag in a `<chassis-inventory>` response tag called `<chassis-inventory>`. Operational responses also can be returned in formatted ASCII, which is enclosed within an `output` element, or in JSON format. For more information about formatting operational responses see [“Specifying the Output Format for Operational Information Requests in a Junos XML Protocol Session” on page 257](#).

The following sample response includes information about the interface ge-2/3/0. The namespace indicated by the `xmlns` attribute in the opening `<interface-information>` tag contains interface information for Junos OS Release 18.1.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.1R1/junos">
  <interface-information xmlns="http://xml.juniper.net/junos/18.1R1/junos-interface">
    <physical-interface>
      <name>ge-2/3/0</name>
      <!-- other data tag elements for the ge-2/3/0 interface -->
    </physical-interface>
  </interface-information>
</rpc-reply>
```

For more information about the `xmlns` attribute and contents of operational response tag elements, see [“Requesting Operational Information Using the Junos XML Protocol” on page 255](#). For a summary of operational response tag elements, see the *Junos XML API Operational Developer Reference*.

Configuration Information Responses

Configuration information responses are responses to requests for information about the device's current configuration. The Junos XML API defines a tag element for every container and leaf statement in the configuration hierarchy. You can instruct the server to return configuration data in different formats including Junos XML elements, formatted ASCII, Junos OS `set` commands, or JSON. If you do not specify a format, the default is XML. For more information about formatting configuration information responses see [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#).

The following sample response includes the information at the `[edit system login]` hierarchy level in the configuration hierarchy. For brevity, the sample shows only one user defined at this level.

```
<rpc-reply xmlns:junos="URL">
  <configuration>
    <system>
      <login>
        <user>
          <name>admin</name>
          <full-name>Administrator</full-name>
          <!-- other data tag elements for the admin user -->
        </user>
      </login>
    </system>
  </configuration>
```

```
</rpc-reply>
```

Configuration Change Responses

Configuration change responses are responses to requests that change the state or contents of the device configuration. For commit operations, the Junos XML protocol server returns the `<commit-results>` response tag, which encloses an explicit indicator of success or failure.

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <!-- tag elements for information about the commit -->
  </commit-results>
</rpc-reply>
```

For other operations, instead of emitting an explicit success indicator, the Junos XML protocol server indicates success by returning an opening `<rpc-reply>` tag and closing `</rpc-reply>` tag with no child elements.

```
<rpc-reply xmlns:junos="URL">
</rpc-reply>
```

For more information, see “Requesting Configuration Changes Using the Junos XML Protocol” on page 160 and “Committing the Candidate Configuration Using the Junos XML Protocol” on page 218. For a summary of the available configuration tag elements, see the *Junos XML API Configuration Developer Reference*.

Related Documentation

- [Understanding the Client Application's Role in a Junos XML Protocol Session on page 53](#)
- [Sending Requests to the Junos XML Protocol Server on page 77](#)
- [Using a Standard API to Parse Response Tag Elements in NETCONF and Junos XML Protocol Sessions on page 82](#)
- [Handling an Error or Warning in Junos XML Protocol Sessions on page 84](#)

Using a Standard API to Parse Response Tag Elements in NETCONF and Junos XML Protocol Sessions

In a NETCONF or Junos XML protocol session, client applications can handle incoming XML tag elements by feeding them to a parser that is based on a standard API such as the Document Object Model (DOM) or Simple API for XML (SAX). Describing how to implement and use a parser is beyond the scope of this documentation

Routines in the DOM accept incoming XML and build a tag hierarchy in the client application's memory. There are also DOM routines for manipulating an existing hierarchy. DOM implementations are available for several programming languages, including C, C++, Perl, and Java. For detailed information, see the *Document Object Model (DOM) Level 1 Specification* from the World Wide Web Consortium (W3C) at <http://www.w3.org/TR/REC-DOM-Level-1/>. Additional information is available from the Comprehensive Perl Archive Network (CPAN) at <http://search.cpan.org/~tjmath/XML-DOM/lib/XML/DOM.pm>.

One potential drawback with DOM is that it always builds a hierarchy of tag elements, which can become very large. If a client application needs to handle only one subhierarchy at a time, it can use a parser that implements SAX instead. SAX accepts XML and feeds the tag elements directly to the client application, which must build its own tag hierarchy. For more information, see the official SAX website at <http://sax.sourceforge.net/>.

- Related Documentation**
- [Parsing the Junos XML Protocol Server Response on page 80](#)
 - [Parsing the NETCONF Server Response](#)

Understanding Character Encoding on Devices Running Junos OS

Junos OS configuration data and operational command output might contain non-ASCII characters, which are outside of the 7-bit ASCII character set. When displaying operational or configuration data in certain formats or within a certain type of session, Junos OS escapes and encodes these characters using the equivalent UTF-8 decimal character reference.

The Junos OS command-line interface (CLI) attempts to display any non-ASCII characters in configuration data that is emitted in text, set, or JSON format, and similarly attempts to display these characters in command output that is emitted in text format. In the exception cases, which include configuration data in XML format and command output in XML or JSON format, the Junos OS CLI displays the UTF-8 decimal character reference instead. In NETCONF and Junos XML protocol sessions, if you request configuration data or command output that contains non-ASCII characters, the server returns the equivalent UTF-8 decimal character reference for those characters for all formats.

For example, suppose the following user account, which contains the Latin small letter *n* with a tilde (*ñ*), is configured on the device running Junos OS.

```
[edit]
user@host# set system login user mariap class super-user uid 2007 full-name "Maria
Peña"
```

When you display the resulting configuration in text format, the CLI prints the corresponding character.

```
[edit]
user@host# show system login user mariap

full-name "Maria Peña";
uid 2007;
class super-user;
```

When you display the resulting configuration in XML format in the CLI or display the configuration in any format in a NETCONF or Junos XML protocol session, the *ñ* character maps to its equivalent UTF-8 decimal character reference `Ã±`.

```
[edit]
user@host# show system login user mariap | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.2R1/junos">
  <configuration junos:changed-seconds="1494033077">
```

```
junos:changed-localtime="2017-05-05 18:11:17 PDT">
  <system>
    <login>
      <user>
        <name>mariap</name>
        <full-name>Maria Pe&#195;&#177;a</full-name>
        <uid>2007</uid>
        <class>super-user</class>
      </user>
    </login>
  </system>
</configuration>
<cli>
  <banner>[edit]</banner>
</cli>
</rpc-reply>
```

When you load configuration data onto a device running Junos OS, you can load non-ASCII characters using their equivalent UTF-8 decimal character reference.

Handling an Error or Warning in Junos XML Protocol Sessions

In a Junos XML protocol session with a device running Junos OS, a client application sends RPCs to the Junos XML protocol server to request information from and manage the configuration on the device. The Junos XML protocol server sends a response to each client request. If the server encounters an error condition, it emits an `<xnm:error>` element containing child elements that describe the error.

The syntax of the `<xnm:error>` element is as follows:

```
<xnm:error xmlns="http://xml.juniper.net/xnm/1.1/xnm" \
  xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
  <!-- tag elements describing the error -->
</xnm:error>
```

The attributes are as follows:

- **xmlns**—The XML namespace for the `<xnm:error>` child tag elements that do not have a prefix in their names (that is, the default namespace for Junos XML tag elements). The value is a URL of the form `http://xml.juniper.net/xnm/version/xnm`, where *version* is a string such as 1.1.
- **xmlns:xnm**—The XML namespace for the `<xnm:error>` tag element and child tag elements that have the `xnm:` prefix in their names. The value is a URL of the form `http://xml.juniper.net/xnm/version/xnm`, where *version* is a string such as 1.1.

The set of child tags enclosed in the `<xnm:error>` element depends on the operation that server was performing when the error occurred. An error can occur while the server is performing any of the following operations, and the server can send a different combination of child tag elements in each case:

- Processing an operational request submitted by a client application
- Opening, locking, changing, committing, or closing a configuration as requested by a client application

- Parsing configuration data submitted by a client application in a `<load-configuration>` tag element

Client applications must be prepared to receive and handle an `<xnm:error>` tag at any time. The information in any response tag elements already received and related to the current request might be incomplete. The client application can include logic for deciding whether to discard or retain the information.

If the Junos XML protocol server encounters a less serious problem, it can emit an `<xnm:warning>` tag element instead. The usual response for the client application in this case is to log the warning or pass it to the user and to continue parsing the server's response.

Related Documentation

- [Sending Requests to the Junos XML Protocol Server on page 77](#)
- [Parsing the Junos XML Protocol Server Response on page 80](#)
- [Halting a Request in Junos XML Protocol Sessions on page 85](#)
- [Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol on page 86](#)
- [<xnm:error> on page 134](#)
- [<xnm:warning> on page 136](#)

Halting a Request in Junos XML Protocol Sessions

In a Junos XML protocol session, to request that the Junos XML protocol server stop processing the current request, a client application emits the `<abort/>` tag directly after the closing `</rpc>` tag for the operation to be halted.

```
<rpc>
  <!-- tag elements for the request -->
</rpc>
<abort/>
```

The Junos XML protocol server responds with the `<abort-acknowledgement/>` tag.

```
<rpc-reply xmlns:junos="URL">
  <abort-acknowledgement/>
</rpc-reply>
```

Depending on the operation being performed, response tag elements already sent by the Junos XML protocol server for the halted request are possibly invalid. The application can include logic for deciding whether to discard or retain them as appropriate.

Related Documentation

- [Sending Requests to the Junos XML Protocol Server on page 77](#)
- [Parsing the Junos XML Protocol Server Response on page 80](#)
- [Handling an Error or Warning in Junos XML Protocol Sessions on page 84](#)
- [<abort/> on page 103](#)
- [<abort-acknowledgement/> on page 125](#)

Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol

When a client application is requesting or changing configuration information, it can use one of the following methods to access the candidate configuration:

- Lock the candidate configuration, which prevents other users or applications from changing the shared configuration database until the application releases the lock (equivalent to the CLI **configure exclusive** command).
- Create a private copy of the candidate configuration, which enables the application to view or change configuration data without affecting the candidate or active configuration until the private copy is committed (equivalent to the CLI **configure private** command).
- Change the candidate configuration without locking it. We do not recommend this method, because of the potential for conflicts with changes made by other applications or users that are editing the shared configuration database at the same time.

If an application is simply requesting configuration information and not changing it, locking the configuration or creating a private copy is not required. The application can begin requesting information immediately. However, if it is important that the information being returned not change during the session, it is appropriate to lock the configuration. The information from a private copy is guaranteed not to change, but can diverge from the candidate configuration if other users or applications are changing the candidate configuration.

The restrictions on, and interactions between, operations on the locked regular candidate configuration and a private copy are the same as for the CLI **configure exclusive** and **configure private** commands. For more information, see [“Committing a Private Copy of the Configuration Using the Junos XML Protocol” on page 219](#) and the *CLI User Guide*.

For more information about locking and unlocking the candidate configuration or creating a private copy, see the following sections:

- [Locking the Candidate Configuration on page 86](#)
- [Unlocking the Candidate Configuration on page 87](#)
- [Creating a Private Copy of the Configuration on page 88](#)

Locking the Candidate Configuration

To lock the candidate configuration, a client application emits the `<lock-configuration/>` tag within an `<rpc>` tag.

```
<rpc>
  <lock-configuration/>
</rpc>
```

Locking the candidate configuration prevents other users or applications from changing the candidate configuration until the lock is released. This is equivalent to the CLI **configure exclusive** command. Locking the configuration before making changes is recommended, particularly on devices where multiple users are authorized to change

the configuration. A commit operation applies to all changes in the candidate configuration, not just those made by the user or application that requests the commit. Allowing multiple users or applications to make changes simultaneously can lead to unexpected results.

The Junos XML protocol confirms that it has locked the candidate configuration by returning an opening `<rpc-reply>` and closing `</rpc-reply>` tag with nothing between them.

```
<rpc-reply xmlns:junos="URL">
</rpc-reply>
```

If the Junos XML protocol server cannot lock the configuration, the `<rpc-reply>` tag instead encloses an `<xnm:error>` element explaining the reason for the failure. Reasons for the failure can include the following:

- Another user or application has already locked the candidate configuration. The error message reports the login identity of the user or application.
- The candidate configuration already includes changes that have not yet been committed. To commit the changes, see [“Committing the Candidate Configuration Using the Junos XML Protocol” on page 218](#). To discard uncommitted changes, see [“Replacing the Candidate Configuration Using the Junos XML Protocol” on page 169](#).

Only one application can hold the lock on the candidate configuration at a time. Other users and applications can read the candidate configuration while it is locked, or can change their private copies. The lock persists until either the Junos XML protocol session ends or the client application unlocks the configuration by emitting the `<unlock-configuration/>` tag, as described in [“Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol” on page 86](#).

If the candidate configuration is not committed before the client application unlocks it, or if the Junos XML protocol session ends for any reason before the changes are committed, the changes are automatically discarded. The candidate and committed configurations remain unchanged.

Unlocking the Candidate Configuration

As long as a client application holds a lock on the candidate configuration, other applications and users cannot change the candidate. To unlock the candidate configuration, the client application includes the `<unlock-configuration/>` tag in an `<rpc>` tag:

```
<rpc>
  <unlock-configuration/>
</rpc>
```

The Junos XML protocol server confirms that it has successfully unlocked the configuration by returning an opening `<rpc-reply>` and closing `</rpc-reply>` tag with nothing between them.

```
<rpc-reply xmlns:junos="URL">
</rpc-reply>
```

If the Junos XML protocol server cannot unlock the configuration, the `<rpc-reply>` tag instead encloses an `<xnm:error>` element explaining the reason for the failure.

Creating a Private Copy of the Configuration

To create a private copy of the candidate configuration, a client application emits the `<private/>` tag enclosed in `<rpc>` and `<open-configuration>` tags.

```
<rpc>
  <open-configuration>
    <private/>
  </open-configuration>
</rpc>
```

The client application can then perform the same operations on the private copy as on the regular candidate configuration.

After making changes to the private copy, the client application can commit the changes to the active configuration on the device running Junos OS by emitting the `<commit-configuration>` tag element, as for the regular candidate configuration. However, there are some restrictions on the commit operation for a private copy. For more information, see “[Committing a Private Copy of the Configuration Using the Junos XML Protocol](#)” on page 219.

To discard the private copy without committing it, a client application emits the `<close-configuration/>` tag enclosed in an `<rpc>` tag element.

```
<rpc>
  <close-configuration/>
</rpc>
```

Any changes to the private copy are lost. Changes to the private copy are also lost if the Junos XML protocol session ends for any reason before the changes are committed. It is not possible to save changes to a private copy other than by emitting the `<commit-configuration>` tag element.

The following example shows how to create a private copy of the configuration. The Junos XML protocol server includes a reminder in its confirmation response that changes are discarded from a private copy if they are not committed before the session ends.

Client Application Junos XML Protocol Server

```
Client Application:
<rpc>
  <open-configuration>
    <private/>
  </open-configuration>
</rpc>

Junos XML Protocol Server:
<rpc-reply xmlns:junos="URL">
  <xnm:warning xmlns="http://xml.juniper.net/xnm/1.1/xnm" \
    xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
    <message>uncommitted changes will be discarded on exit</message>
  </xnm:warning>
</rpc-reply>
```

T1172

- Related Documentation**
- [Understanding the Client Application's Role in a Junos XML Protocol Session on page 53](#)
 - [Sending Requests to the Junos XML Protocol Server on page 77](#)

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [<lock-configuration/> on page 118](#)
- [<unlock-configuration/> on page 122](#)
- [<open-configuration> on page 119](#)

Terminating Junos XML Protocol Sessions

In a Junos XML protocol session, a client application's attempt to lock the candidate configuration can fail because another user or application already holds the lock. In this case, the Junos XML protocol server returns an error message that includes the username and process ID (PID) for the entity that holds the existing lock:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <xnm:error>
    <message>
      configuration database locked by:
      user terminal (pid PID) on since YYYY-MM-DD hh:mm:ss TZ, idle hh:mm:ss
      exclusive [edit]
    </message>
  </xnm:error>
</rpc-reply>
```

If the client application has the Junos OS **maintenance** permission, it can end the session that holds the lock by emitting the **<kill-session>** and **<session-id>** tag elements in an **<rpc>** element. The **<session-id>** element specifies the PID obtained from the error message:

```
<rpc>
  <kill-session>
    <session-id>PID</session-id>
  </kill-session>
</rpc>
```

The Junos XML protocol server confirms that it has terminated the other session by returning the **<ok/>** tag in the **<rpc-reply>** tag element:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
```

We recommend that the application include logic for determining whether it is appropriate to terminate another session, based on factors such as the identity of the user or application that holds the lock, or the length of idle time.

When a session is terminated, the Junos XML protocol server that is servicing the session rolls back all uncommitted changes that have been made during the session. If a confirmed commit is pending (changes have been committed but not yet confirmed), the Junos XML protocol server restores the configuration to its state before the confirmed commit instruction was issued. For information about the confirmed commit operation, see [“Committing the Candidate Configuration Only After Confirmation Using the Junos XML Protocol” on page 223](#).

- Related Documentation**
- [Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol on page 86](#)
 - [Sending Requests to the Junos XML Protocol Server on page 77](#)
 - [Ending a Junos XML Protocol Session and Closing the Connection on page 90](#)
 - [Handling an Error or Warning in Junos XML Protocol Sessions on page 84](#)
 - [<kill-session> on page 114](#)

Ending a Junos XML Protocol Session and Closing the Connection

In a Junos XML protocol session with a device running Junos OS, when a client application is finished making requests, it ends the session by emitting the `<request-end-session/>` tag within an `<rpc>` tag element:

```
<rpc>
  <request-end-session/>
</rpc>
```

In response, the Junos XML protocol server emits the `<end-session/>` tag enclosed in an `<rpc-reply>` tag element and a closing `</junoscript>` tag:

```
<rpc-reply xmlns:junos="URL">
  <end-session/>
</rpc-reply>
</junoscript>
```

The client application waits to receive this reply before emitting its closing `</junoscript>` tag:

```
</junoscript>
```

The client application can then close the SSH, SSL, or other connection to the Junos XML protocol server device.

Similarly, client applications written in Perl can end the Junos XML protocol session and close the connection by using the Junos XML protocol Perl module and methods described in [“Closing the Connection to the Junos XML Protocol Server in Perl Client Applications” on page 367](#). Client applications that do not use the Junos XML protocol Perl module use the routine provided for closing a connection in the standard library for their programming language.

- Related Documentation**
- [Understanding the Client Application’s Role in a Junos XML Protocol Session on page 53](#)
 - [Starting Junos XML Protocol Sessions on page 70](#)
 - [Sending Requests to the Junos XML Protocol Server on page 77](#)
 - [Terminating Junos XML Protocol Sessions on page 89](#)
 - [<request-end-session/> on page 120](#)

Sample Junos XML Protocol Session

The following sections describe the sequence of tag elements in a sample Junos XML protocol session with a device running Junos OS. The client application begins by establishing a connection to a Junos XML protocol server.

- [Exchanging Initialization PIs and Tag Elements on page 91](#)
- [Sending an Operational Request on page 91](#)
- [Locking the Configuration on page 92](#)
- [Changing the Configuration on page 92](#)
- [Committing the Configuration on page 93](#)
- [Unlocking the Configuration on page 93](#)
- [Closing the Junos XML Protocol Session on page 94](#)

Exchanging Initialization PIs and Tag Elements

After the client application establishes a connection to a Junos XML protocol server, the two exchange initialization PIs and tag elements, as shown in the following example. Note that the Junos XML protocol server's opening `<junoscript>` tag appears on multiple lines for legibility only. Neither the Junos XML protocol server nor the client application inserts a newline character into the list of attributes. Also, in an actual exchange, the *JUNOS-release* variable is replaced by a value such as 18.1R1 for Junos OS Release 18.1. For a detailed discussion of the `<?xml?>` PI and opening `<junoscript>` tag, see ["Starting Junos XML Protocol Sessions" on page 70](#).

Client Application

```
<?xml version="1.0" encoding="us-ascii"?>
<junoscript version="1.0" release="JUNOS-release">
```

Junos XML Protocol Server

```
<?xml version="1.0" encoding="us-ascii"?>
<junoscript version="1.0" hostname="router1" \
  os="JUNOS" release="JUNOS-release" \
  xmlns="URL"xmlns:junos="URL" \
  xmlns:xnm="URL">
```

T1173

Sending an Operational Request

The client application emits the `<get-chassis-inventory>` tag element to request information about the device's chassis hardware. The Junos XML protocol server returns the requested information in the `<chassis-inventory>` tag element.

Client Application	Junos XML Protocol Server
<pre> <rpc> <get-chassis-inventory> <detail/> </get-chassis-inventory> </rpc> </pre>	<pre> <rpc-reply xmlns:junos="URL"> <chassis-inventory xmlns="URL"> <chassis> <name>Chassis</name> <serial-number>1122</serial-number> <description>M320</description> <chassis-module> <name>Midplane</name> <!-- other child tags for the Midplane --> </chassis-module> <!-- tags for other chassis modules --> </chassis> </chassis-inventory> </rpc-reply> </pre>

T1102

Locking the Configuration

The client application then prepares to incorporate a change into the candidate configuration by emitting the `<lock-configuration/>` tag to prevent any other users or applications from altering the candidate configuration at the same time. To confirm that the candidate configuration is locked, the Junos XML protocol server returns only an opening `<rpc-reply>` tag and a closing `</rpc-reply>` tag with no child elements. For more information about locking the configuration, see [“Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol”](#) on page 86.

Client Application	Junos XML Protocol Server
<pre> <rpc> <lock-configuration/> </rpc> </pre>	<pre> <rpc-reply xmlns:junos="URL"> </rpc-reply> </pre>

T1103

Changing the Configuration

The client application now emits tag elements to create a new Junos OS login class called **network-mgmt** at the `[edit system login class]` hierarchy level in the candidate configuration. The Junos XML protocol server returns the `<load-configuration-results>` tag, which encloses a child element that reports the outcome of the load operation. (Understanding the meaning of these tag elements is not necessary for the purposes of this example, but for information about them, see [“Requesting Configuration Changes Using the Junos XML Protocol”](#) on page 160.)

Client Application

```

<rpc>
  <load-configuration>
    <configuration>
      <system>
        <login>
          <class>
            <name>network-mgmt</name>
            <permissions>configure</permissions>
            <permissions>snmp</permissions>
            <permissions>system</permissions>
          </class>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1104

Committing the Configuration

The client application then commits the candidate configuration. The Junos XML protocol server returns the **<commit-results>** tag, which encloses child elements that report the outcome of the commit operation.

Client Application

```

<rpc>
  <commit-configuration/>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re0</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>

```

T1105

Unlocking the Configuration

The client application unlocks (and by implication closes) the candidate configuration. To confirm that the unlock operation was successful, the Junos XML protocol server returns only an opening **<rpc-reply>** tag and a closing **</rpc-reply>** tag with no child elements.

Client Application

```

<rpc>
  <unlock-configuration/>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
</rpc-reply>

```

T1106

Closing the Junos XML Protocol Session

The client application closes the Junos XML protocol session by emitting the `<request-end-session>` tag.

Client Application	Junos XML Protocol Server
<code><rpc></code>	
<code> <request-end-session/></code>	
<code></rpc></code>	<code><rpc-reply xmlns:junos="URL"></code>
	<code> <end-session/></code>
	<code></rpc-reply></code>
	<code></junoscript></code>
<code></junoscript></code>	

T1165

- Related Documentation
- [Understanding the Client Application's Role in a Junos XML Protocol Session on page 53](#)
 - [Connecting to the Junos XML Protocol Server on page 68](#)
 - [Starting Junos XML Protocol Sessions on page 70](#)
 - [Sending Requests to the Junos XML Protocol Server on page 77](#)
 - [Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol on page 86](#)
 - [Ending a Junos XML Protocol Session and Closing the Connection on page 90](#)

CHAPTER 6

Junos XML Protocol Tracing Operations

- [NETCONF and Junos XML Protocol Tracing Operations Overview on page 95](#)
- [Example: Tracing NETCONF and Junos XML Protocol Session Operations on page 96](#)

NETCONF and Junos XML Protocol Tracing Operations Overview

You can configure tracing operations for the NETCONF and Junos XML management protocols. NETCONF and Junos XML protocol tracing operations record NETCONF and Junos XML protocol session data, respectively, in a trace file. By default, NETCONF and Junos XML protocol tracing operations are not enabled.



NOTE: Starting in Junos OS Release 16.1, when you enable tracing operations at the `[edit system services netconf traceoptions]` hierarchy, Junos OS enables tracing operations for both NETCONF and Junos XML protocol sessions and adds the `[NETCONF]` and `[JUNOScript]` tags to the log file entries to distinguish the type of session. Prior to Junos OS Release 16.1, only NETCONF session data was logged, and the `[NETCONF]` tag was omitted.

You configure NETCONF and Junos XML protocol tracing operations at the `[edit system services netconf traceoptions]` hierarchy level.

```
[edit system services]
netconf {
  traceoptions {
    file <filename> <files number> <match regular-expression> <size size>
      <world-readable | no-world-readable>;
    flag flag;
    no-remote-trace;
    on-demand;
  }
}
```

To enable NETCONF and Junos XML protocol tracing operations and to trace all incoming and outgoing data from NETCONF and Junos XML protocol sessions on that device, configure the **flag all** statement. As of Junos OS Release 16.1, a new option under the **flag** statement, **debug**, is introduced. This option enables debug-level tracing. However, we recommend using the **flag all** option. You can restrict tracing to only incoming or outgoing NETCONF or Junos XML protocol data by configuring the flag value as either **incoming**

or **outgoing**, respectively. Additionally, to restrict the trace output to include only those lines that match a particular expression, configure the **file match** statement and define the regular expression against which the output is matched.

NETCONF and Junos XML protocol tracing operations record session data in the file `/var/log/netconf`. To specify a different trace file, configure the **file** statement and desired filename.

By default, when the trace file reaches 128 KB in size, it is renamed and compressed to **filename.0.gz**, then **filename.1.gz**, and so on, until there are 10 trace files. Then the oldest trace file (**filename.9.gz**) is overwritten. You can configure limits on the number and size of trace files by including the **file files number** and **file size size** statements. You can configure up to a maximum of 1000 files. Specify the file size in bytes or use **sizek** to specify KB, **sizem** to specify MB, or **sizeg** to specify GB. You cannot configure the maximum number of trace files and the maximum trace file size independently. If one option is configured, the other option must also be configured along with a filename.

To control the tracing operation from within a NETCONF or Junos XML protocol session, configure the **on-demand** statement. This requires that you start and stop tracing operations from within the session. If you configure the **on-demand** statement, you must issue the `<rpc><request-netconf-trace><start/></request-netconf-trace></rpc>` RPC in the session to start tracing operations for that session. To stop tracing for that session, issue the `<rpc><request-netconf-trace><stop/></request-netconf-trace></rpc>` RPC.

By default, access to the trace file is restricted to the owner. You can manually configure access by including either the **world-readable** or **no-world-readable** statement. The **no-world-readable** statement restricts trace file access to the owner. This is the default. The **world-readable** statement enables unrestricted access to the trace file.

Release History Table

Release	Description
16.1	Starting in Junos OS Release 16.1, when you enable tracing operations at the [edit system services netconf traceoptions] hierarchy, Junos OS enables tracing operations for both NETCONF and Junos XML protocol sessions and adds the [NETCONF] and [JUNOScript] tags to the log file entries to distinguish the type of session.

Related Documentation

- [Example: Tracing NETCONF and Junos XML Protocol Session Operations on page 96](#)
- [netconf](#)
- [ssh \(NETCONF\)](#)
- [traceoptions \(NETCONF and Junos XML Protocol\) on page 385](#)

Example: Tracing NETCONF and Junos XML Protocol Session Operations

This example demonstrates how to configure tracing operations for NETCONF and Junos XML protocol sessions.



NOTE: Starting in Junos OS Release 16.1, when you enable tracing operations at the `[edit system services netconf traceoptions]` hierarchy, Junos OS enables tracing operations for both NETCONF and Junos XML protocol sessions and adds the `[NETCONF]` and `[JUNOScript]` tags to the log file entries to distinguish the type of session. Prior to Junos OS Release 16.1, only NETCONF session data was logged, and the `[NETCONF]` tag was omitted.

- [Requirements on page 97](#)
- [Overview on page 97](#)
- [Configuration on page 97](#)
- [Verification on page 99](#)

Requirements

- A routing, switching, or security device running Junos OS Release 16.1 or later is required.

Overview

This example configures basic tracing operations for NETCONF and Junos XML protocol sessions. The example configures the trace file **netconf-ops.log** and sets a maximum number of 20 trace files and a maximum size of 3 MB for each file. The **flag all** statement configures tracing for all incoming and outgoing NETCONF or Junos XML protocol data. The **world-readable** option enables unrestricted access to the trace files.

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 services netconf ssh
set system services netconf traceoptions file netconf-ops.log
set system services netconf traceoptions file size 3m
set system services netconf traceoptions file files 20
set system services netconf traceoptions file world-readable
set system services netconf traceoptions flag all
```

Configuring NETCONF and Junos XML Protocol Tracing Operations

Step-by-Step Procedure

To configure NETCONF and Junos XML protocol tracing operations:

1. For NETCONF sessions, enable NETCONF over SSH.


```
[edit]
user@R1# set system services netconf ssh
```
2. Configure the traceoptions flag to specify which session data to capture.

You can specify incoming, outgoing, or all data. This example configures tracing for all session data.

```
[edit]
user@R1# set system services netconf traceoptions flag all
```

3. (Optional) Configure the filename of the trace file.

The following statement configures the trace file **netconf-ops.log**, which is stored in the **/var/log** directory. If you do not specify a filename, NETCONF and Junos XML protocol session data is stored in **/var/log/netconf**.

```
[edit]
user@R1# set system services netconf traceoptions file netconf-ops.log
```

4. (Optional) Configure the maximum number of trace files and the maximum size of each file.

The following statements configure a maximum of 20 trace files with a maximum size of 3 MB per file.

```
[edit]
user@R1# set system services netconf traceoptions file files 20
user@R1# set system services netconf traceoptions file size 3m
```

5. (Optional) Restrict the trace output to include only those lines that match a particular regular expression.

The following configuration, which is not used in this example, matches on and logs only session data that contains "error-message".

```
[edit]
user@R1# set system services netconf traceoptions file match error-message
```

6. (Optional) Configure on-demand tracing to control tracing operations from the NETCONF or Junos XML protocol session.

The following configuration, which is not used in this example, enables on-demand tracing.

```
[edit]
user@R1# set system services netconf traceoptions on-demand
```

7. (Optional) Configure the permissions on the trace file by specifying whether the file is **world-readable** or **no-world-readable**.

This example enables unrestricted access to the trace file.

```
[edit]
user@R1# set system services netconf traceoptions file world-readable
```

8. Commit the configuration.

```
[edit]
user@R1# commit
```

Results

```
[edit]
system {
  services {
    netconf {
      ssh;
      traceoptions {
        file netconf-ops.log size 3m files 20 world-readable;
        flag all;
      }
    }
  }
}
```

Verification

Verifying NETCONF and Junos XML protocol Tracing Operation

Purpose Verify that the device is writing NETCONF and Junos XML protocol session data to the configured trace file. This example logs both incoming and outgoing NETCONF and Junos XML protocol data. In the sample NETCONF session, which is not detailed here, the user modifies the candidate configuration on R1 to include the **bgp-troubleshoot.slax** op script and then commits the configuration.

Action Display the trace output of the configured trace file `/var/log/netconf-ops.log` by issuing the **show log** operational mode command.

```

user@R1 show log netconf-ops.log
Apr  3 13:09:04 [NETCONF] Started tracing session: 3694
Apr  3 13:09:29 [NETCONF] - [3694] Incoming: <rpc>
Apr  3 13:09:29 [NETCONF] - [3694] Outgoing: <rpc-reply
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/16.1R1/junos">
Apr  3 13:09:39 [NETCONF] - [3694] Incoming: <edit-config>
Apr  3 13:09:43 [NETCONF] - [3694] Incoming: <target>
Apr  3 13:09:47 [NETCONF] - [3694] Incoming: <candidate/>
Apr  3 13:09:53 [NETCONF] - [3694] Incoming: </target>
Apr  3 13:10:07 [NETCONF] - [3694] Incoming:
<default-operation>merge</default-operation>
Apr  3 13:10:10 [NETCONF] - [3694] Incoming: <config>
Apr  3 13:10:13 [NETCONF] - [3694] Incoming: <configuration>
Apr  3 13:10:16 [NETCONF] - [3694] Incoming: <system>
Apr  3 13:10:19 [NETCONF] - [3694] Incoming: <scripts>
Apr  3 13:10:23 [NETCONF] - [3694] Incoming: <op>
Apr  3 13:10:26 [NETCONF] - [3694] Incoming: <file>
Apr  3 13:10:44 [NETCONF] - [3694] Incoming: <name>bgp-troubleshoot.slax</name>
Apr  3 13:10:46 [NETCONF] - [3694] Incoming: </file>
Apr  3 13:10:48 [NETCONF] - [3694] Incoming: </op>
Apr  3 13:10:52 [NETCONF] - [3694] Incoming: </scripts>
Apr  3 13:10:56 [NETCONF] - [3694] Incoming: </system>
Apr  3 13:11:00 [NETCONF] - [3694] Incoming: </configuration>
Apr  3 13:11:00 [NETCONF] - [3694] Outgoing: <ok/>
Apr  3 13:11:12 [NETCONF] - [3694] Incoming: </config>
Apr  3 13:11:18 [NETCONF] - [3694] Incoming: </edit-config>
Apr  3 13:11:26 [NETCONF] - [3694] Incoming: </rpc>
Apr  3 13:11:26 [NETCONF] - [3694] Outgoing: </rpc-reply>
Apr  3 13:11:26 [NETCONF] - [3694] Outgoing: ]]>]]>
Apr  3 13:11:31 [NETCONF] - [3694] Incoming: ]]>]]>

Apr  3 13:14:20 [NETCONF] - [3694] Incoming: <rpc>
Apr  3 13:14:20 [NETCONF] - [3694] Outgoing: <rpc-reply
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/16.1R1/junos">
Apr  3 13:14:26 [NETCONF] - [3694] Incoming: <commit/>
Apr  3 13:14:35 [NETCONF] - [3694] Outgoing: <ok/>
Apr  3 13:14:35 [NETCONF] - [3694] Incoming: </rpc>
Apr  3 13:14:35 [NETCONF] - [3694] Outgoing: </rpc-reply>
Apr  3 13:14:35 [NETCONF] - [3694] Outgoing: ]]>]]>
Apr  3 13:14:40 [NETCONF] - [3694] Incoming: ]]>]]>

Apr  3 13:30:48 [NETCONF] - [3694] Outgoing: <!-- session end at 2016-12-03
13:30:48 PDT -->

```

Meaning This example configured the **flag all** statement, so the trace file displays all incoming and outgoing NETCONF or Junos XML protocol session operations. Each operation includes the date and timestamp. The log file indicates the type of session, either NETCONF or Junos XML protocol, by including the **[NETCONF]** or **[JUNOScript]** tag, respectively. Multiple NETCONF and Junos XML protocol sessions are distinguished by a session number. In this example, only one NETCONF session, using session identifier 3694, is active.

Release History Table

Release	Description
16.1	Starting in Junos OS Release 16.1, when you enable tracing operations at the [edit system services netconf traceoptions] hierarchy, Junos OS enables tracing operations for both NETCONF and Junos XML protocol sessions and adds the [NETCONF] and [JUNOScript] tags to the log file entries to distinguish the type of session. Prior to Junos OS Release 16.1, only NETCONF session data was logged, and the [NETCONF] tag was omitted.

Related Documentation

- [NETCONF and Junos XML Protocol Tracing Operations Overview on page 95](#)
- *netconf*
- *ssh (NETCONF)*
- [traceoptions \(NETCONF and Junos XML Protocol\) on page 385](#)

CHAPTER 7

Junos XML Protocol Operations

<abort/>

Usage	<pre><rpc> <!-- child tag elements --> </rpc> <abort/></pre>
Release Information	This is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.
Description	Direct the NETCONF or Junos XML protocol server to stop processing the request that is currently outstanding. The server responds by returning the <abort-acknowledgment/> tag, but might already have sent tagged data in response to the request. The client application must discard those tag elements.
Related Documentation	<ul style="list-style-type: none">• Halting a Request in Junos XML Protocol Sessions on page 85• <abort-acknowledgment/> on page 125

<close-configuration/>

Usage	<pre><rpc> <close-configuration/> </rpc></pre>
Release Information	This is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.
Description	<p>Close the open configuration database and discard any uncommitted changes.</p> <p>This tag element is normally used to close a private copy of the candidate configuration or an open instance of the ephemeral configuration database and discard any</p>

uncommitted changes. The application must have previously emitted the `<open-configuration>` tag element. Closing the NETCONF or Junos XML protocol session (by emitting the `<request-end-session/>` tag, for example) has the same effect as emitting this tag element.

- Related Documentation**
- [Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol on page 86](#)
 - [<open-configuration> on page 119](#)
 - [<request-end-session/> on page 120](#)

`<commit-configuration>`

Usage	<pre><rpc> <commit-configuration/> <commit-configuration> <check/> </commit-configuration> <commit-configuration> <log>log-message</log> </commit-configuration> <commit-configuration> <at-time>time-specification</at-time> <log>log-message</log> </commit-configuration> <commit-configuration> <confirmed/> <confirm-timeout>rollback-delay</confirm-timeout> <log>log-message</log> </commit-configuration> <commit-configuration> <synchronize/> <log>log-message</log> </commit-configuration> <commit-configuration> <synchronize/> <at-time>time-specification</at-time> <log>log-message</log> </commit-configuration> <commit-configuration> <synchronize/> <check/> <log>log-message</log> </commit-configuration> <commit-configuration> <synchronize/> <confirmed/> <confirm-timeout>rollback-delay</confirm-timeout></pre>
-------	--


```

        <log>log-message</log>
    </commit-configuration>

    <commit-configuration>
        <synchronize/>
        <force-synchronize/>
    </commit-configuration>
</rpc>

```

Release Information This is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI <http://xml.juniper.net/netconf/junos/1.0> in the capabilities exchange.

Description Request that the NETCONF or Junos XML protocol server perform one of the variants of the commit operation on the candidate configuration, a private copy of the candidate configuration, or an open instance of the ephemeral configuration database.

Some restrictions apply to the commit operation for a private copy of the candidate configuration and for the ephemeral configuration database. For example, the commit operation fails for a private copy if the regular candidate configuration is locked by another user or application or if it includes uncommitted changes made since the private copy was created. Also, a commit operation on an instance of the ephemeral configuration database only supports the `<synchronize/>` option.

Enclose the appropriate tag in the `<commit-configuration>` tag element to specify the type of commit operation:

- To commit the configuration immediately, making it the active configuration on the device, emit the empty `<commit-configuration/>` tag.
- To verify the syntactic correctness of the candidate configuration or a private copy without actually committing it, enclose the `<check/>` tag in the `<commit-configuration>` tag element.
- To record a message in the commit history log when the associated commit operation succeeds, define the log message string in the `<log>` tag element and enclose the tag element in the `<commit-configuration>` tag element. The `<log>` tag element can be combined with any other tag element. When the `<log>` tag element is emitted alone, the associated commit operation begins immediately.

- To commit the candidate configuration but roll back to the previous configuration after a short time, enclose the `<confirmed/>` tag in the `<commit-configuration>` tag element.

By default, the rollback occurs after 10 minutes; to set a different rollback delay, also emit the optional `<confirm-timeout>` tag element. To delay the rollback again (past the original rollback deadline), emit the `<confirmed/>` tag (enclosed in the `<commit-configuration>` tag element) before the deadline passes. Include the `<confirm-timeout>` tag element to specify how long to delay the next rollback, or omit that tag element to use the default of 10 minutes. The rollback can be delayed repeatedly in this way.

To commit the configuration immediately and permanently after emitting the `<confirmed/>` tag, emit the empty `<commit-configuration/>` tag before the rollback deadline passes. The device commits the candidate configuration and cancels the rollback. If the candidate configuration is still the same as the current committed configuration, the effect is the same as recommitting the current committed configuration.



NOTE: The confirmed commit operation is not available when committing a private copy of the configuration or an open instance of the ephemeral configuration database.

- On a device with two Routing Engines, commit the candidate configuration, private copy, or ephemeral database instance stored on the local Routing Engine on both Routing Engines. Combine tag elements as indicated in the following (the ephemeral database only supports the `<synchronize/>` option):
 - To copy the candidate configuration or the configuration data in the open ephemeral instance that is stored on the local Routing Engine to the other Routing Engine, verify the configuration's syntactic correctness, and commit it immediately on both Routing Engines, enclose the `<synchronize/>` tag in the `<commit-configuration>` tag element.
 - To copy the candidate configuration stored on the local Routing Engine to the other Routing Engine, verify the candidate's syntactic correctness, and commit it on both Routing Engines at a defined future time, enclose the `<synchronize/>` or `<force-synchronize/>` tag and `<at-time>` tag element in the `<commit-configuration>` tag element. Set the value in the `<at-time>` tag element as previously described for use of the `<at-time>` tag element alone.
 - To copy the candidate configuration stored on the local Routing Engine to the other Routing Engine and verify the candidate's syntactic correctness on each Routing Engine, enclose the `<synchronize/>` or `<force-synchronize/>` and `<check/>` tag elements in the `<commit-configuration>` tag element.
 - To copy the candidate configuration stored on the local Routing Engine to the other Routing Engine, verify the candidate's syntactic correctness, and commit it on both Routing Engines but require confirmation, enclose the `<synchronize/>` tag and `<confirmed/>` tag elements, and optionally the `<confirm-timeout>` tag element, in the `<commit-configuration>` tag element. Set the value in the `<confirm-timeout>`

tag element as previously described for use of the `<confirmed/>` tag and `<confirm-timeout>` tag element alone.

- To force the same synchronized commit operation as invoked by the `<synchronize/>` tag to succeed, even if there are open configuration sessions or uncommitted configuration changes on the remote machine, enclose the `<force-synchronize/>` tag in the `<commit-configuration>` tag element.
- To schedule the candidate configuration for commit at a future time, enclose the `<at-time>` tag element in the `<commit-configuration>` tag element. There are three valid types of time specifiers:
 - The string **reboot**, to commit the configuration the next time the device reboots.
 - A time value of the form *hh:mm[:ss]* (hours, minutes, and, optionally, seconds), to commit the configuration at the specified time, which must be in the future but before 11:59:59 PM on the day the `<commit-configuration>` tag element is emitted. Use 24-hour time for the *hh* value; for example, 04:30:00 means 4:30:00 AM and 20:00 means 8:00 PM. The time is interpreted with respect to the clock and time zone settings on the device.
 - A date and time value of the form *yyyy-mm-dd hh:mm[:ss]* (year, month, date, hours, minutes, and, optionally, seconds), to commit the configuration at the specified date and time, which must be after the `<commit-configuration>` tag element is emitted. Use 24-hour time for the *hh* value. For example, 2005-08-21 15:30:00 means 3:30 PM on August 21, 2005. The time is interpreted with respect to the clock and time zone settings on the device.



NOTE: The time you specify must be more than 1 minute later than the current time on the device.

The configuration is checked immediately for syntactic correctness. If the check succeeds, the configuration is scheduled for commit at the specified time. If the check fails, the commit operation is not scheduled.

- Contents**
- `<at-time>`—Schedule the commit operation for a specified future time.
 - `<check>`—Request verification that the configuration is syntactically correct, but do not actually commit it.
 - `<confirmed>`—Request a commit of the candidate configuration and a rollback to the previous configuration after a short time, 10 minutes by default. Use the `<confirm-timeout>` tag element to specify a different amount of time.
 - `<confirm-timeout>`—Specify the number of minutes for which the configuration remains active when the `<confirmed/>` tag is enclosed in the `<commit-configuration>` tag element.
 - `<log>`—Record a message in the commit history log when the commit operation succeeds.

<synchronize>—On dual control plane systems, request that the configuration on one control plane be copied to the other control plane, checked for correct syntax, and committed on both Routing Engines.

<force-synchronize>—On dual control plane systems, force the candidate configuration on one control plane to be copied to the other control plane.

Related Documentation

- [Committing the Candidate Configuration Using the Junos XML Protocol on page 218](#)
- [Committing a Private Copy of the Configuration Using the Junos XML Protocol on page 219](#)
- [Committing a Configuration at a Specified Time Using the Junos XML Protocol on page 221](#)
- [Committing the Candidate Configuration Only After Confirmation Using the Junos XML Protocol on page 223](#)
- [Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol on page 225](#)
- [Logging a Message About a Commit Operation Using the Junos XML Protocol on page 233](#)
- [<commit-results> on page 127](#)
- [<open-configuration> on page 119](#)

<get-checksum-information>

Usage

```
<rpc>  
  <get-checksum-information>  
    <path>  
    <!-- name and path of file -->  
  </path>  
  </get-checksum-information>  
</rpc>
```

Release Information This is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI `http://xml.juniper.net/netconf/junos/1.0` in the capabilities exchange.

Description Request checksum information for the specified file.

Contents **<path>**—Name and path of the file to check.

Usage Guidelines See the *Junos XML API Operational Developer Reference*.

Related Documentation

- [<checksum-information> on page 127](#)

<get-configuration>

```

Usage      <rpc>
           <get-configuration
             [changed="changed"]
             [commit-scripts="( apply | apply-no-transients | view )"]
             [compare="rollback" [rollback="[0-49]"]]
             [database="(candidate | committed)"]
             [database-path=$junos-context/commit-context/database-path]
             [format="( json | set | text | xml )"]
             [inherit="( defaults | inherit )"
               [groups="groups" [interface-ranges="interface-ranges"]]
             [(junos:key | key )="key"] >

           <!-- tag elements for the configuration element to display -->
           </get-configuration>
         </rpc>

```

Release Information This is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI `http://xml.juniper.net/netconf/junos/1.0` in the capabilities exchange.

commit-scripts attribute values **apply** and **apply-no-transients** added in Junos OS Release 12.1

database-path attribute added in Junos OS Release 12.2.

format attribute value **json** added in Junos OS Release 14.2.

action attribute value **set** added in Junos OS Release 15.1.

Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.

Starting in Junos OS Releases 16.1R4, 16.2R2, and 17.1R1, integers in Junos OS configuration data emitted in JSON format are not enclosed in quotation marks.

Description Request configuration data from the NETCONF or Junos XML protocol server. The attributes specify the source and formatting of the data to display.

If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing the **<get-configuration>** operation, the server returns the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration, unless the active configuration is explicitly requested by including the **database="committed"** attribute.

A client application can request the entire configuration hierarchy or a section of it.

- To display the entire configuration hierarchy, emit the empty **<get-configuration/>** tag.
- To display a configuration element (hierarchy level or configuration object), emit tag elements within the **<get-configuration>** tag element to represent all levels of the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to the level or object to display. To represent a hierarchy level or a configuration object that does not have an identifier, emit it as an empty tag. To represent an object

that has one or more identifiers, emit its container tag element and identifier tag elements only, not any tag elements that represent other characteristics.



NOTE: To retrieve configuration data from an instance of the ephemeral configuration database, a client application must first open the ephemeral instance using the `<open-configuration>` operation with the appropriate child tags before emitting the `<get-configuration>` operation. When retrieving ephemeral configuration data using the `<get-configuration>` operation, the only supported attributes are `format` and `key`.



NOTE: Starting in Junos OS Release 13.1, within a NETCONF or Junos XML protocol session, a logical system user can use the Junos XML `<get-configuration>` operation to request specific logical system configuration hierarchies using child configuration tags as well as request the entire logical system configuration. When requesting the entire logical system configuration, the RPC reply includes the `<configuration>` root tag. Prior to Junos OS Release 13.1, the `<configuration>` root tag is omitted.

Attributes **changed**—Specify that the `junos:changed="changed"` attribute should appear in the opening tag of each changed configuration element.

The attribute appears in the opening tag of every parent tag element in the path to the changed configuration element, including the top-level opening `<configuration>` tag. If the changed configuration element is represented by a single (empty) tag, the `junos:changed="changed"` attribute appears in the tag. If the changed element is represented by a container tag element, the `junos:changed="changed"` attribute appears in the opening container tag and also in each child tag element enclosed in the container tag element.

The **database** attribute can be combined with the `changed="changed"` attribute to request either the candidate or active configuration:

- When the candidate configuration is requested (the `database="candidate"` attribute is included or the **database** attribute is omitted completely), elements added to the candidate configuration after the last commit operation are marked with the `junos:changed="changed"` attribute.
- When the active configuration is requested (the `database="committed"` attribute is included), elements added to the active configuration by the most recent commit are marked with the `junos:changed="changed"` attribute.



NOTE: When a commit operation succeeds, the server removes the `junos:changed="changed"` attribute from all tag elements. However, if warnings are generated during the commit, the attribute is not removed. In this case, the `junos:changed="changed"` attribute appears in tag elements that changed before the commit operation as well as on those that changed after it.

An example of a commit-time warning is the message explaining that a configuration element will not actually apply until the device is rebooted. The warning appears in the tag string that the server returns to confirm the success of the commit, enclosed in an `<xnm:warning>` tag element.

To remove the `junos:changed="changed"` attribute from elements that changed before the commit, take the action necessary to eliminate the cause of the warning, and commit the configuration again.

commit-scripts—Request that the NETCONF or Junos XML protocol server display commit-script-style XML data. The value of the attribute determines the output. Acceptable values are:

- **apply**—Display the configuration with commit script changes applied, including both transient and non-transient changes. The output is equivalent to the CLI output when using the `| display commit-scripts` option.
- **apply-no-transients**—Display the configuration with commit script changes applied, but exclude transient changes. The output is equivalent to the CLI output when using the `| display commit-scripts no-transients` option.
- **view**—Display the configuration in the XML format that is input to a commit script. This is equivalent to viewing the configuration with the attributes `inherit="inherit"`, `groups="groups"`, and `changed="changed"`. The output is equivalent to the CLI output when using the `| display commit-scripts view` option.

compare—Request that the NETCONF or Junos XML protocol server display the differences between the active or candidate configuration and a previously committed configuration. The only acceptable value for the **compare** attribute is **rollback**. The compare attribute is combined with the `rollback="rollback-number"` to specify which previously committed configuration should be used in the comparison. If the **rollback** attribute is omitted, the comparison uses rollback number 0, which is the active configuration.

The **database** attribute can be combined with the `compare="rollback"` attribute to request either the candidate or active configuration. If the **database** attribute is omitted, the candidate configuration is used. When the **compare** attribute is used, the default format for the output is text. If the client application attempts to include the `format="xml"` attribute when the `compare="rollback"` attribute is present, the server will return an `<xnm:error>` element indicating an error.

database—Specify the version of the configuration from which to display data. There are two acceptable values:

- **candidate**—The candidate configuration.
- **committed**—The active configuration (the one most recently committed).

The **database** attribute takes precedence over the **database-path** attribute, if both are included.

database-path—Within a commit script, this attribute specifies the path to the session's pre-inheritance candidate configuration. The only acceptable value is **\$junos-context/commit-context/database-path**.

For normal configuration sessions, the commit script retrieves the normal, pre-inheritance candidate configuration. For private configuration sessions, the commit script retrieves the private, pre-inheritance candidate configuration.

If you include both the **database** and the **database-path** attributes, the **database** attribute takes precedence.

format—Specify the format in which the NETCONF or Junos XML protocol server returns the configuration data. Acceptable values are:

- **json**—Configuration statements are formatted using JavaScript Object Notation (JSON). Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.



NOTE: Starting in Junos OS Releases 16.1R4, 16.2R2, and 17.1R1, integers in Junos OS configuration data emitted in JSON format are not enclosed in quotation marks. In earlier releases, integers in JSON configuration data were treated as strings and enclosed in quotation marks.

- **set**—Configuration statements are formatted as Junos OS configuration mode **set** commands.
- **text**—Configuration statements are formatted as ASCII text, using the newline character, tabs and other white space, braces, and square brackets to indicate the hierarchical relationships between the statements. This is the format used in configuration files stored on a device running Junos OS and displayed by the CLI **show configuration** command.
- **xml**—Configuration statements are represented by the corresponding Junos XML tag elements. This is the default value if the **format** attribute is omitted.

groups—Specify that the **junos:group="group-name"** attribute appear in the opening tag for each configuration element that is inherited from a configuration group. The

group-name variable specifies the name of the configuration group from which that element was inherited.

The **groups** attribute must be combined with the **inherit** attribute, and the one acceptable value for it is **groups**.

inherit—Specify how the NETCONF or Junos XML protocol server display statements that are defined in configuration groups and interface ranges. If the **inherit** attribute is omitted, the output uses the **<groups>**, **<apply-groups>**, and **<apply-groups-except>** tag elements to represent user-defined configuration groups and uses the **<interface-range>** tag element to represent user-defined interface ranges; it does not include tag elements for statements defined in the junos-defaults group.

The acceptable values are:

- **defaults**—The output does not include the **<groups>**, **<apply-groups>**, and **<apply-groups-except>** tag elements, but instead displays tag elements that are inherited from user-defined groups and from the junos-defaults group as children of the inheriting tag elements.
- **inherit**—The output does not include the **<groups>**, **<apply-groups>**, **<apply-groups-except>**, and **<interface-range>** tag elements, but instead displays tag elements that are inherited from user-defined groups and ranges as children of the inheriting tag elements. The output does not include tag elements for statements defined in the junos-defaults group.

interface-ranges—Specify that the **junos:interface-ranges="source-interface-range"** attribute appear in the opening tag for each configuration element that is inherited from an interface range. The *source-interface-range* variable specifies the name of the interface range.

The **interface-ranges** attribute must be combined with the **inherit** attribute, and the one acceptable value for it is **interface-ranges**.

junos:key | key—Specify that the **junos:key="key"** attribute appear in the opening tag of each element that serves as an identifier for a configuration object. The only acceptable value is **key**.

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [junos:changed on page 143](#)
- [junos:group on page 146](#)
- [junos:interface-range on page 146](#)
- [junos:key on page 147](#)
- *Junos XML API Configuration Developer Reference*

<kill-session>

Usage	<pre><rpc> <kill-session> <session-id>PID</session-id> </kill-session> </rpc></pre>
Description	<p>Request that the Junos XML protocol server terminate another CLI or Junos XML protocol session. The usual reason to emit this tag is that the user or application for the other session holds a lock on the candidate configuration, preventing the client application from locking the configuration itself.</p> <p>The client application must have the Junos OS maintenance permission to perform this operation.</p>
Contents	<p><session-id>—PID of the entity conducting the session to terminate. The PID is reported in the <xnm:error> tag element that the Junos XML protocol server generates when it cannot lock a configuration as requested.</p>
Related Documentation	<ul style="list-style-type: none">• Terminating Junos XML Protocol Sessions on page 89• <lock-configuration/> on page 118• <xnm:error> on page 134

<load-configuration>

Usage	<pre><rpc> <load-configuration rescue="rescue"/> <load-configuration rollback="index"/> <load-configuration url="url" [action="(merge override replace update)"] [format="(text xml)"] /> <load-configuration url="url" [action="(merge override update)"] format="json" /> <load-configuration url="url" action="set" format="text"/> <load-configuration [action="(merge override replace update)"] [format="xml"]> <configuration> <!-- tag elements for configuration elements to load --> </configuration> </load-configuration> <load-configuration [action="(merge override replace update)"] format="text"> <configuration-text></pre>
-------	--

```

        <!-- formatted ASCII configuration statements to load -->
    </configuration-text>
</load-configuration>

<load-configuration [action="(merge | override | update)"] format="json">
    <configuration-json>
        <!-- JSON configuration data to load -->
    </configuration-json>
</load-configuration>

<load-configuration action="set" format="text">
    <configuration-set>
        <!-- configuration mode commands to load -->
    </configuration-set>
</load-configuration>
</rpc>

```

Release Information This is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI <http://xml.juniper.net/netconf/junos/1.0> in the capabilities exchange.

action attribute value **set** added in Junos OS Release 11.4.

format attribute value **json** added in Junos OS Release 16.1.

Description Request that the NETCONF or Junos XML protocol server load configuration data into the candidate configuration or open configuration database.

If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing the **<load-configuration>** operation, the server loads the configuration data into the open configuration database. Otherwise, the server loads the configuration data into the candidate configuration.

Provide the data to load in one of the following ways:

- Set the empty **<load-configuration/>** tag's **rescue** attribute to the value **rescue**. The rescue configuration completely replaces the candidate configuration.
- Set the empty **<load-configuration/>** tag's **rollback** attribute to the numerical index of a previous configuration. The device stores a copy of the most recently committed configuration and up to 49 previous configurations. The specified previous configuration completely replaces the candidate configuration.
- Set the empty **<load-configuration/>** tag's **url** attribute to the pathname of a file that contains the configuration data to load. If providing the configuration data as formatted ASCII text, set the **format** attribute to **text**. If providing the configuration data as Junos XML tag elements, either omit the **format** attribute or set the value to **xml**. If providing the configuration data as configuration mode commands, set the **action** attribute to **set**, and either omit the **format** attribute or set the value to **text**. If providing the configuration data in JavaScript Object Notation (JSON), set the **format** attribute to **json**.

In the following example, the `url` attribute identifies that the configuration data should be loaded from the `/tmp/add.conf` file.

```
<load-configuration url="/tmp/add.conf"/>
```

- Enclose the configuration data as a data stream within an opening `<load-configuration>` and closing `</load-configuration>` tag. If providing the configuration data as formatted ASCII text, enclose the data in a `<configuration-text>` tag element, and set the `format` attribute to `text`. If providing the configuration data as Junos XML tag elements, enclose the data in a `<configuration>` tag element, and either omit the `format` attribute or set the value to `xml`. If providing the configuration data as configuration mode commands, enclose the data in a `<configuration-set>` tag element, set the `action` attribute to `set`, and either omit the `format` attribute or set the value to `text`. If providing the configuration data in JSON, enclose the data in a `<configuration-json>` tag element, and set the `format` attribute to `json`.

Attributes **action**—Specify how to load the configuration data, particularly when the target configuration database and the loaded configuration contain conflicting statements.



NOTE: The ephemeral configuration database only supports the `action` attribute values of `"merge"` and `"set"`.

The following are acceptable values:

- **merge**—Combine the data in the loaded configuration with the data in the target configuration. If statements in the loaded configuration conflict with statements in the target configuration, the loaded statements replace those in the target configuration. This is the default behavior if the `action` attribute is omitted.
- **override**—Discard the entire candidate configuration and replace it with the loaded configuration. When the configuration is later committed, all system processes parse the new configuration.
- **replace**—Substitute each hierarchy level or configuration object defined in the loaded configuration for the corresponding level or object in the candidate configuration.

If providing the configuration data as formatted ASCII text (either in the file named by the `url` attribute or enclosed in a `<configuration-text>` tag element), also place the **replace**: statement on the line directly preceding the statements that represent the hierarchy level or object to replace. For more information, see the discussion of loading a file of configuration data in the *CLI User Guide*.

If providing the configuration data as Junos XML tag elements, include the **replace="replace"** attribute in the opening tags of the elements that represent the hierarchy levels or objects to replace.

- **set**—Load configuration data formatted as Junos OS configuration mode commands. This option executes the configuration instructions line by line as they

are stored in a file named by the **url** attribute or enclosed in a **<configuration-set>** tag element. The instructions can contain any configuration mode command, such as **set**, **delete**, **edit**, or **deactivate**. When providing the configuration data as a set of commands, the only acceptable value for the **format** attribute is "text". If the **action** attribute value is "set", and the **format** attribute is omitted, the **format** attribute automatically defaults to "text" rather than **xml**.

- **update**—Compare a complete loaded configuration against the candidate configuration. For each hierarchy level or configuration object that is different in the two configurations, the version in the loaded configuration replaces the version in the candidate configuration. When the configuration is later committed, only system processes that are affected by the changed configuration elements parse the new configuration.

format—Specify the format used for the configuration data. Acceptable values are:

- **json**—Indicate that the configuration data is formatted using JavaScript Object Notation (JSON).
- **text**—Indicate that the configuration data is formatted as ASCII text or as a set of configuration mode commands.

ASCII text format uses the newline character, tabs and other white space, braces, and square brackets to indicate the hierarchical relationships between the statements. This is the format used in configuration files stored on the routing platform and is the format displayed by the CLI **show configuration** command. The **set** command format consists of a series of Junos OS configuration mode commands and is displayed by the **show configuration | display set** CLI command. To load a set of configuration mode commands, you must set the **action** attribute to "set".

- **xml**—Indicate that the configuration data is formatted using Junos XML tag elements. If the **format** attribute is omitted, "xml" is the default format for all values of the **action** attribute except "set", which defaults to format "text".

rescue—Specify that the rescue configuration replace the current candidate configuration. The only valid value is "rescue".



NOTE: If an application does not support executing RPCs with XML attributes, starting in Junos OS Release 17.4R2 you can also use the **<rollback-config>** RPC to load the rescue configuration.

rollback—Specify the numerical index of the previous configuration to load. Valid values are 0 (zero, for the most recently committed configuration) through one less than the number of stored previous configurations (maximum is 49).



NOTE: If an application does not support executing RPCs with XML attributes, starting in Junos OS Release 17.4R2 you can also use the `<rollback-config>` RPC to load a previously committed configuration.

url—Specify the full pathname of the file that contains the configuration data to load. The value can be a local file path, an FTP location, or a Hypertext Transfer Protocol (HTTP) URL:

- A local filename can have one of the following forms:
 - `/path/filename`—File on a mounted file system, either on the local flash disk or on hard disk.
 - `a:filename` or `a:path/filename`—File on the local drive. The default path is `/` (the root-level directory). The removable media can be in MS-DOS or UNIX (UFS) format.

- A filename on an FTP server has the following form:

`ftp://username:password@hostname/path/filename`

- A filename on an HTTP server has the following form:

`http://username:password@hostname/path/filename`

In each case, the default value for the *path* variable is the home directory for the username. To specify an absolute path, the application starts the path with the characters `%2F`; for example, `ftp://username:password@hostname/%2Fpath/filename`.

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [<load-configuration-results> on page 132](#)
- [replace on page 152](#)
- entries for `<configuration>`, `<configuration-text>`, and `<configuration-set>` in the *Junos XML API Configuration Developer Reference*

<lock-configuration/>

Usage

```
<rpc>
  <lock-configuration/>
</rpc>
```

Release Information

This is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI `http://xml.juniper.net/netconf/junos/1.0` in the capabilities exchange.

Description Request that the NETCONF or Junos XML protocol server open and lock the candidate configuration, enabling the client application both to read and change it, but preventing any other users or applications from changing it. The application must emit the `<unlock-configuration/>` tag to unlock the configuration.

If the Junos XML protocol session ends or the application emits the `<unlock-configuration/>` tag before the candidate configuration is committed, all changes made to the candidate are discarded.

- Related Documentation**
- [Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol on page 86](#)
 - `<unlock-configuration/>` on page 122

`<open-configuration>`

Usage

```
<rpc>
  <open-configuration>
    <private/>
  </open-configuration>

  <open-configuration>
    <ephemeral/>
  </open-configuration>

  <open-configuration>
    <ephemeral-instance>instance-name</ephemeral-instance>
  </open-configuration>
</rpc>
```

Release Information This is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI `http://xml.juniper.net/netconf/junos/1.0` in the capabilities exchange.

`<ephemeral>` and `<ephemeral-instance>` elements added in Junos OS Release 16.2R2.

Description Create a private copy of the candidate configuration or open the default instance or a user-defined instance of the ephemeral configuration database.



NOTE: Before opening a user-defined instance of the ephemeral configuration database, you must first enable the instance by configuring the instance *instance-name* statement at the `[edit system configuration-database ephemeral]` hierarchy level on the device.

A client application can perform the same operations on the private copy or ephemeral instance as on the regular candidate configuration, including load and commit operations.

There are, however, restrictions on these operations. For details, see [“<load-configuration>” on page 114](#) and [“<commit-configuration>” on page 104](#).

To close a private copy or ephemeral instance and discard all uncommitted changes, emit the empty `<close-configuration/>` tag in an `<rpc>` element. Changes to the private copy or ephemeral instance are also lost if the NETCONF or Junos XML protocol session ends for any reason before the changes are committed. It is not possible to save the changes other than by performing a commit operation, for example, by emitting the `<commit-configuration/>` tag.

Contents

- `<private/>`—Open a private copy of the candidate configuration.
- `<ephemeral/>`—Open the default instance of the ephemeral configuration database.
- `<ephemeral-instance>`—Open the specified instance of the ephemeral configuration database. This instance must already be configured at the `[edit system configuration-database ephemeral]` hierarchy level on the device.

Related Documentation

- [Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol on page 86](#).
- [<close-configuration/> on page 103](#)
- [<commit-configuration> on page 104](#)
- [<lock-configuration/> on page 118](#)

`<request-end-session/>`

Usage

```
<rpc>
  <request-end-session/>
</rpc>
```

Release Information This is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI `http://xml.juniper.net/netconf/junos/1.0` in the capabilities exchange.

Description Request that the NETCONF or Junos XML protocol server end the current session.

Related Documentation

- [<end-session/> on page 131](#)

`<request-login>`

Usage

```
<rpc>
  <request-login>
    <username>account</username>
    <challenge-response>password</challenge-response>
```



```
</request-login>
</rpc>
```

Release Information XML special characters in the username or password elements must be escaped starting in Junos OS Releases 13.3R7, 14.1R6, 14.2R4, 15.1R2, and 16.1R1.

Description Request authentication by the Junos XML protocol server when using the clear-text or SSL access protocol.

Emitting both the **<username>** and **<challenge-response>** tag elements is appropriate if the client application automates access to device information and does not interact with users, or obtains the password from a user before beginning the authentication process.

Emitting only the **<username>** tag element is appropriate if the application does not obtain the password until the authentication process has already begun. In this case, the Junos XML protocol server returns the **<challenge>** tag element to request the password associated with the account.



NOTE: Starting in Junos OS Releases 13.3R7, 14.1R6, 14.2R4, 15.1R2, and 16.1R1, any XML special characters in the username or password elements of a **<request-login>** RPC request must be escaped. The following five symbols are considered special characters: greater than (>), less than (<), single quote ('), double quote ("), and ampersand (&). Both entity references and character references are acceptable escape sequence formats. For example, **&** and **&** are valid representations of an ampersand.

Contents **<challenge-response>**—Specify the password for the account named in the **<username>** tag element. Omit this tag element to have the Junos XML protocol server emit the **<challenge>** tag element to request the password.

<username>—Name of the user account under which to authenticate with the Junos XML protocol server. The account must already be configured on the device where the Junos XML protocol server is running.

Related Documentation

- [Authenticating with the Junos XML Protocol Server for Cleartext or SSL Connections on page 75](#)
- [<challenge> on page 126](#)
- [<rpc> on page 121](#)

<rpc>

Usage

```
<junoscript>
  <rpc [attributes]>
    <!-- tag elements in a request from a client application -->
  </rpc>
```

</junoscript>

Description Enclose all tag elements in a request generated by a client application.

Attributes (Optional) One or more attributes of the form *attribute-name="value"*. This feature can be used to associate requests and responses if the value assigned to an attribute by the client application is unique in each opening <rpc> tag. The Junos XML protocol server echoes the attribute unchanged in its opening <rpc-reply> tag, making it simple to map the response to the initiating request.



NOTE: The xmlns:junos attribute name is reserved. The Junos XML protocol server sets the attribute to an appropriate value on the opening <rpc-reply> tag, so client applications must not emit it on the opening <rpc> tag.

Related Documentation

- [Sending Requests to the Junos XML Protocol Server on page 77](#)
- [<junoscript> on page 123](#)
- [<rpc-reply> on page 134](#)

<unlock-configuration/>

Usage

```
<rpc>  
  <unlock-configuration/>  
</rpc>
```

Release Information This is a Junos XML management protocol operation. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI `http://xml.juniper.net/netconf/junos/1.0` in the capabilities exchange.

Description Request that the NETCONF or Junos XML protocol server unlock and close the candidate configuration. Until the application emits this tag, other users or applications can read the configuration but cannot change it.

Related Documentation

- [Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol on page 86](#)
- [<lock-configuration/> on page 118](#)

CHAPTER 8

Junos XML Protocol Processing Instructions

<?xml?>

Usage	<code><?xml version="version" encoding="encoding"?></code>
Description	Specify the XML version and character encoding scheme for the session.
Attributes	encoding —Specify the standardized character set that the emitter uses and can interpret. version —Specify the version of XML used by the emitter.
Related Documentation	<ul style="list-style-type: none">• Starting Junos XML Protocol Sessions on page 70• <junoscript> on page 123

<junoscript>

Usage	<pre><!-- emitted by a client application --> <junoscript version="version" [hostname="hostname"] [junos:key="key"] [release="release"]> <!-- all tag elements generated by the application during the session --> </junoscript> <!-- emitted by the Junos XML protocol server --> <junoscript xmlns="namespace-URL" xmlns:junos="namespace-URL" schemaLocation="namespace-URL" os="os" release="release" hostname="hostname" version="version"> <!-- all tag elements generated by the Junos XML protocol server during the session --> </junoscript></pre>
Description	Enclose all tag elements in a Junos XML protocol session (act as the root tag element for the session). The client application and Junos XML protocol server each emit this tag element, enclosing all other tag elements that they emit during a session inside it.

Attributes **hostname**—Name of the device on which the tag element's originator is running.

junos:key—Request that the Junos XML protocol server include the **junos:key="key"** attribute in the opening tag of each tag element that serves as an identifier for a configuration object.

os—Operating system of the device named by the **hostname** attribute.

release—Identify the Junos OS release being run by the tag element's originator. Software modules always set this attribute, but client applications are not required to set it.

schemaLocation—XML namespace for the XML Schema-language representation of the Junos OS configuration hierarchy.

version—(Required for a client application) Specify the version of the Junos XML management protocol used for the enclosed set of tag elements.

xmlns—XML namespace for the tag elements enclosed by the **<junoscript>** tag element that do not have a prefix on their names (that is, the default namespace for Junos XML tag elements). The value is a URL of the form **http://xml.juniper.net/xnm/version-code/xnm**, where *version-code* is a string such as 1.1.

xmlns:junos—XML namespace for the tag elements enclosed by the **<junoscript>** tag element that have the **junos:** prefix. The value is a URL of the form **http://xml.juniper.net/junos/release-code/junos**, where *release-code* is the standard string that represents a release of the Junos OS, such as 18.1R1 for the initial version of Junos OS Release 18.1.

**Related
Documentation**

- [Starting Junos XML Protocol Sessions on page 70](#)
- [Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol on page 295](#)
- [<rpc> on page 121](#)
- [<rpc-reply> on page 134](#)
- [junos:key on page 147](#)

CHAPTER 9

Junos XML Protocol Response Tags

<abort-acknowledgement/>

Usage

```
<rpc-reply xmlns:junos="URL">
  <any-child-of-rpc-reply>
    <abort-acknowledgement/>
  </any-child-of-rpc-reply>
</rpc-reply>
```

Release Information This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI <http://xml.juniper.net/netconf/junos/1.0> in the capabilities exchange.

Description Indicates that the NETCONF or Junos XML protocol server has received the **<abort/>** tag and has stopped processing the current request. If the client application receives any tag elements related to the request between sending the **<abort/>** tag and receiving this tag, it must discard them.

Related Documentation

- [<xnm:error> on page 134](#)

<authentication-response>

Usage

```
<rpc-reply xmlns:junos="URL">
  <authentication-response>
    <status>authentication-outcome</status>
    <message>message</message>
    <login-name>remote-username</login-name>
  </authentication-response>
</rpc-reply>
```

Description Indicate whether an authentication attempt succeeded. The Junos XML protocol server returns the tag element in response to the **<request-login>** tag element emitted by a client application that uses the clear-text or Secure Sockets Layer (SSL) access protocol.

Contents	<login-name> —Specifies the username that the client application provided to an authentication utility such as RADIUS or TACACS+. This tag element appears only if the username that it contains differs from the username contained in the <message> tag element.
	<message> —Names the account under which a connection to the Junos XML protocol server is established, if authentication succeeds. If authentication fails, explains the reason for the failure.
	<status> —Indicates whether the authentication attempt succeeded. There are two possible values: <ul style="list-style-type: none">• fail—The attempt failed. The Junos XML protocol server also emits the <challenge> tag element to request the password again, up to a maximum of three attempts.• success—The attempt succeeded. An authenticated connection to the Junos XML protocol server is established.
Related Documentation	<ul style="list-style-type: none">• Authenticating with the Junos XML Protocol Server for Cleartext or SSL Connections on page 75• <challenge> on page 126• <request-login> on page 120• <rpc-reply> on page 134

<challenge>

Usage	<pre><rpc-reply xmlns:junos="URL"> <challenge echo="no">Password:</challenge> </rpc-reply></pre>
Description	<p>Request the password associated with an account during authentication with a client application that uses the clear-text or SSL access protocol. The Junos XML protocol server emits this tag element when the initial <request-login> tag element emitted by the client application does not enclose a <challenge-response> tag element, and when the password enclosed in a <challenge-response> tag element is incorrect (in the latter case, the server also emits an <authentication-response> tag element enclosing child tag elements that indicate the password is incorrect).</p> <p>The tag element encloses the string Password: which the client application can forward to the screen as a prompt for a user.</p>
Attributes	<p>echo—Specifies whether the password string typed by the user appears on the screen. The value "no" specifies that it does not.</p>

- | | |
|------------------------------|--|
| Related Documentation | <ul style="list-style-type: none"> • Authenticating with the Junos XML Protocol Server for Cleartext or SSL Connections on page 75 • Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server on page 57 • <authentication-response> on page 125 • <request-login> on page 120 • <rpc-reply> on page 134 |
|------------------------------|--|

<checksum-information>

Usage	<pre> <rpc-reply> <checksum-information> <file-checksum> <computation-method>MD5</computation-method> <input-file> <!-- name and path of file--> </input-file> </file-checksum> </checksum-information> </rpc-reply> </pre>
Release Information	<p>This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI <code>http://xml.juniper.net/netconf/junos/1.0</code> in the capabilities exchange.</p>
Description	<p>Encloses tag elements that include the file to check, the checksum algorithm used, and the checksum output.</p>
Contents	<p><checksum>—Resulting value from the checksum computation.</p> <p><computation-method>—Checksum algorithm used. Currently, all checksum computations use the MD5 algorithm; thus, the only possible value is MD5.</p> <p><file-checksum>—Wrapper that holds the resulting <input-file>, <computation-method>, and <checksum> attributes for a particular checksum computation.</p> <p><input-file>—Name and path of the file that the checksum algorithm was run against.</p>
Related Documentation	<ul style="list-style-type: none"> • <get-checksum-information> on page 108

<commit-results>

Usage	<pre> <rpc-reply xmlns:junos="URL"> <!-- for the candidate configuration or ephemeral configuration --> <commit-results> </pre>
--------------	---

```
        <routing-engine>...</routing-engine>
    </commit-results>

    <!-- for a private copy -->
    <commit-results>
        <load-success/>
        <routing-engine>...</routing-engine>
    </commit-results>

    <!-- for a private copy that does not include changes -->
    <commit-results>
    </commit-results>
</rpc-reply>
```

Release Information	This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.
Description	Tag element returned by the Junos XML protocol server in response to a <code><commit-configuration></code> request by a client application. The <code><commit-results></code> element contains information about the requested commit operation performed by the server on a particular Routing Engine.
Contents	<code><load-success/></code> —Indicates that the Junos XML protocol server successfully merged changes from the private copy into a copy of the candidate configuration, before committing the combined candidate on the specified Routing Engine. The <code><routing-engine></code> tag element is described separately.
Related Documentation	<ul style="list-style-type: none">• Committing the Candidate Configuration Using the Junos XML Protocol on page 218• <commit-configuration> on page 104• <routing-engine> on page 133

<commit-revision-information>

Usage	<pre><rpc-reply xmlns:junos="URL"> <commit-results> <routing-engine> <!-- configuration with commit revision identifier --> <commit-revision-information> <old-db-revision>old-revision-id</old-db-revision> <new-db-revision>new-revision-id</new-db-revision> </commit-revision-information> </routing-engine> </commit-results> </rpc-reply></pre>
--------------	---

Release Information	This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI <code>http://xml.juniper.net/netconf/junos/1.0</code> in the capabilities exchange. Element introduced in Junos OS Release 16.1.
Description	<p>Child element included in a Junos XML protocol server <commit-results> response element to return information about the old and new configuration request identifiers on a particular Routing Engine. The configuration request identifier is used by network management server (NMS) applications, such as Junos Space, to determine whether the synchronization (sync) status of a device that the NMS application manages is out-of-sync or in-sync.</p> <p>Out-of-band configuration changes are configuration changes made to a device outside of the network management server (NMS) application, such as Junos Space. For example, configuration changes can be performed on a device using the device CLI, using the device Web-based management interface (the J-Web interface or Web View), or using the Junos Space Network Management Platform configuration editor. As a result, there is a requirement for a configuration revision identifier to determine whether the configuration settings on devices being managed by an NMS application is in sync with the CLI of devices running Junos OS. A configuration revision identifier might not be necessary if the NMS application is the only utility that is used to modify the configuration of a device. However, in a real-world network deployment, out-of-band configuration commits might occur on a device, such as during a maintenance window for support operations. In such cases, the NMS application might not detect these out-of-band commits. To solve this problem, starting in Junos OS Release 16.1, the <commit-revision-information> element containing a configuration revision identifier string is enclosed within the <commit-results> and <routing-engine> tags. The configuration revision identifier is a string (for example, <code>re0-1365168149-1</code>), which has the following format:</p> <pre><routing-engine-name>-<timestamp>-<counter></pre>
Contents	<p><old-db-revision>—Indicates the old configuration revision identifier, which is the identifier of the configuration prior to the previously successfully committed configuration.</p> <p><new-db-revision>—Indicates the new configuration revision identifier, which is the identifier of the last successfully committed configuration.</p> <p><revision-id>—Unique identifier of the revision made to the configuration in the database, which contains the name of the Routing Engine on which the commit operation was performed.</p>
Related Documentation	<ul style="list-style-type: none"> • <commit-results> on page 127 • <routing-engine> on page 133

<database-status>

Usage `<junoscript>`

```

<any-child-of-junoscript>
  <xnm:error>
    <database-status-information>
      <database-status>
        <user>username</user>
        <terminal>terminal</terminal>
        <pid>pid</pid>
        <start-time>start-time</start-time>
        <idle-time>idle-time</idle-time>
        <commit-at>time</commit-at>
        <exclusive/>
        <edit-path>edit-path</edit-path>
      </database-status>
    </database-status-information>
  </xnm:error>
</any-child-of-junoscript>
</junoscript>

```

Description Describe a user or Junos XML protocol client application that is logged in to the configuration database. For simplicity, the Contents section uses the term user to refer to both human users and client applications, except where the information differs for the two.

Contents

- <commit-at/>**—Indicates that the user has scheduled a commit operation for a later time.
- <edit-path>**—Specifies the user's current location in the configuration hierarchy, in the form of the CLI configuration mode banner.
- <exclusive/>**—Indicates that the user or application has an exclusive lock on the configuration database. A user enters exclusive configuration mode by issuing the **configure exclusive** command in CLI operational mode. A client application obtains the lock by emitting the **<lock-configuration/>** tag element.
- <idle-time>**—Specifies how much time has passed since the user last performed an operation in the database.
- <pid>**—Specifies the process ID of the Junos management process (mgd) that is handling the user's login session.
- <start-time>**—Specifies the time when the user logged in to the configuration database, in the format **YYYY-MM-DD hh:mm:ss TZ** (year, month, date, hour in 24-hour format, minute, second, time zone).
- <terminal>**—Identifies the UNIX terminal assigned to the user's connection.
- <user>**—Specifies the Junos OS login ID of the user whose login to the configuration database caused the error.

Related Documentation

- [Handling an Error or Warning in Junos XML Protocol Sessions on page 84](#)
- [<database-status-information> on page 131](#)
- [<junoscript> on page 123](#)

- [<xnm:error> on page 134](#)

<database-status-information>

Usage	<pre> <junoscript> <any-child-of-junoscript> <xnm:error> <database-status-information> <database-status>...</database-status> </database-status-information> <xnm:error> </any-child-of-junoscript> </junoscript> </pre>
Description	<p>Describe one or more users who have an open editing session in the configuration database.</p> <p>The <database-status> tag element is explained separately.</p>
Related Documentation	<ul style="list-style-type: none"> • Handling an Error or Warning in Junos XML Protocol Sessions on page 84 • <database-status> on page 129 • <junoscript> on page 123 • <xnm:error> on page 134

<end-session/>

Usage	<pre> <rpc-reply xmlns:junos="URL"> <end-session/> </rpc-reply> </pre>
Release Information	<p>This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.</p>
Description	<p>Indicates that the NETCONF or Junos XML protocol server is about to end the current session for a reason other than an error. Most often, the reason is that the client application has sent the <request-end-session/> tag.</p>
Related Documentation	<ul style="list-style-type: none"> • Ending a Junos XML Protocol Session and Closing the Connection on page 90 • <request-end-session/> on page 120

<load-configuration-results>

Usage	<pre><rpc-reply xmlns:junos="URL"> <load-configuration-results> <load-success/> <load-error-count>errors</load-error-count> </load-configuration-results> </rpc-reply></pre>
Release Information	This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI http://xml.juniper.net/netconf/junos/1.0 in the capabilities exchange.
Description	<p>Tag element returned by the NETCONF or Junos XML protocol server in response to a <load-configuration> request by a client application.</p> <p>In a Junos XML protocol session, the <load-configuration-results> element encloses either a <load-success/> tag or a <load-error-count> tag, which indicates the success or failure of the load configuration operation. In a NETCONF session, the <load-configuration-results> element encloses either an <ok/> tag or a <load-error-count> tag to indicate the success or failure of the load configuration operation.</p>
Contents	<p><load-error-count>—Specifies the number of errors that occurred when the server attempted to load new data into the candidate configuration or open configuration database. The target configuration must be restored to a valid state before it is committed.</p> <p><load-success/>—Indicates that the server successfully loaded new data into the candidate configuration or open configuration database.</p>
Related Documentation	<ul style="list-style-type: none">• <load-configuration> on page 114

<reason>

Usage	<pre><xnm:error xnm:warning> <reason> <daemon>process</daemon> <process-not-configured/> <process-disabled/> <process-not-running/> </reason> </xnm:error xnm:warning></pre>
Description	Child element included in an <xnm:error> or <xnm:warning> element in a Junos XML protocol server response to explain why a process could not service a request.

- Contents**
- <daemon>**—Identifies the process.
 - <process-disabled>**—Indicates that the process has been explicitly disabled by an administrator.
 - <process-not-configured>**—Indicates that the process has been disabled because it is not configured.
 - <process-not-running>**—Indicates that the process is not running.

- Related Documentation**
- [Handling an Error or Warning in Junos XML Protocol Sessions on page 84](#)
 - [<xnm:error> on page 134](#)
 - [<xnm:warning> on page 136](#)

<routing-engine>

Usage

```
<rpc-reply xmlns:junos="URL">
  <commit-results>

    <!-- when the candidate configuration or private copy is committed -->
    <routing-engine>
      <name>reX</name>
      <commit-success/>
      <commit-revision-information>
        <old-db-revision>old-revision-id</old-db-revision>
        <new-db-revision>new-revision-id</new-db-revision>
      </commit-revision-information>
    </routing-engine>

    <!-- when the candidate configuration or private copy is syntactically
    valid -->
    <routing-engine>
      <name>reX</name>
      <commit-check-success/>
    </routing-engine>

    <!-- when an instance of the ephemeral database is committed -->
    <routing-engine>
      <name>reX</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

Release Information This is a Junos XML management protocol response tag. It is supported in Junos XML protocol sessions, and it is supported as a Juniper Networks proprietary extension in NETCONF sessions on devices running Junos OS that identify the URI <http://xml.juniper.net/netconf/junos/1.0> in the capabilities exchange.

Description Child element included in a Junos XML protocol server **<commit-results>** response element to return information about a requested commit operation on a particular Routing Engine.

- Contents**
- <commit-check-success>**—Indicates that the configuration is syntactically correct.
 - <commit-success>**—Indicates that the Junos XML protocol server successfully committed the configuration.
 - <name>**—Name of the Routing Engine on which the commit operation was performed. Possible values are re0 and re1.
- The **<commit-revision-information>** tag element is described separately.

- Related Documentation**
- [<commit-results> on page 127](#)
 - [<commit-revision-information> on page 128](#)

<rpc-reply>

- Usage**
- ```
<junoscript>
 <rpc-reply xmlns:junos="namespace-URL">
 <!-- tag elements in a reply from the Junos XML protocol server -->
 </rpc-reply>
</junoscript>
```
- Description**
- Enclose all tag elements in a reply from the Junos XML protocol server. The immediate child tag element is usually one of the following:
- The tag element used to enclose data generated by the Junos XML protocol server or a Junos OS module in response to a client application's request.
  - The **<output>** tag element, if the Junos XML API does not define a specific tag element for the requested information.
- Attributes**
- xmlns:junos**—Names the XML namespace for the Junos XML tag elements enclosed by the **<rpc-reply>** tag element that have the **junos:** prefix. The value is a URL of the form **http://xml.juniper.net/junos/release-code/junos**, where *release-code* is the standard string that represents a Junos OS release, such as 18.1R1 for the initial version of Junos OS Release 18.1.
- Related Documentation**
- [Parsing the Junos XML Protocol Server Response on page 80](#)
  - [<junoscript> on page 123](#)
  - **<output>** in the *Junos XML API Operational Developer Reference*
  - [<rpc> on page 121](#)

---

## <xnm:error>

- Usage**
- ```
<junoscript>
  <any-child-of-junoscript>
    <xnm:error xmlns="namespace-URL" xmlns:xnm="namespace-URL">
```

```

    <parse/>
    <source-daemon>module-name </source-daemon>
    <filename>filename</filename>
    <line-number>line-number </line-number>
    <column>column-number</column>
    <token>input-token-id </token>
    <edit-path>edit-path</edit-path>
    <statement>statement-name </statement>
    <message>error-string</message>
    <re-name>re-name-string</re-name>
    <database-status-information>...</database-status-information>
    <reason>...</reason>
  </xnm:error>
</any-child-of-junoscript>
</junoscript>

```

Description Indicate that the Junos XML protocol server has experienced an error while processing the client application's request. If the server has already emitted the response tag element for the current request, the information enclosed in the response tag element might be incomplete. The client application must include code that discards or retains the information, as appropriate. The child tag elements described in the Contents section detail the nature of the error. The Junos XML protocol server does not necessarily emit all child tag elements; it omits tag elements that are not relevant to the current request.

Attributes **xmlns**—XML namespace for the contents of the tag element. The value is a URL of the form `http://xml.juniper.net/xnm/version/xnm`, where *version* is a string such as "1.1".

xmlns:xnm—XML namespace for child tag elements that have the **xnm:** prefix on their names. The value is a URL of the form `http://xml.juniper.net/xnm/version/xnm`, where *version* is a string such as "1.1".

Contents **<column>**—(Occurs only during loading of a configuration file) Identifies the element that caused the error by specifying its position as the number of characters after the first character in the specified line in the configuration file that was being loaded. The line and file are specified by the accompanying **<line-number>** and **<filename>** tag elements.

<edit-path>—(Occurs only during loading of configuration data) Specifies the path to the configuration hierarchy level at which the error occurred, in the form of the CLI configuration mode banner.

<filename>—(Occurs only during loading of a configuration file) Names the configuration file that was being loaded.

<line-number>—(Occurs only during loading of a configuration file) Specifies the line number where the error occurred in the configuration file that was being loaded, which is named by the accompanying **<filename>** tag element.

<message>—Describes the error in a natural-language text string.

<parse/>—Indicates that there was a syntactic error in the request submitted by the client application.

<re-name>—Names the Routing Engine on which the error occurred.

<source-daemon>—Names the Junos OS module that was processing the request in which the error occurred.

<statement>—(Occurs only during loading of configuration data) Identifies the configuration statement that was being processed when the error occurred. The accompanying **<edit-path>** tag element specifies the statement's parent hierarchy level.

<token>—Names which element in the request caused the error.

The other tag elements are explained separately.

Related Documentation

- [Handling an Error or Warning in Junos XML Protocol Sessions on page 84](#)
- [<database-status-information> on page 131](#)
- [<junoscript> on page 123](#)
- [<reason> on page 132](#)
- [<xnm:warning> on page 136](#)

<xnm:warning>

Usage

```
<junoscript>
  <any-child-of-junoscript>
    <xnm:warning xmlns="namespace-URL" xmlns:xnm="namespace-URL">
      <source-daemon>module-name </source-daemon>
      <filename>filename</filename>
      <line-number>line-number </line-number>
      <column>column-number</column>
      <token>input-token-id </token>
      <edit-path>edit-path</edit-path>
      <statement>statement-name </statement>
      <message>error-string</message>
      <reason>...</reason>
    </xnm:warning>
  </any-child-of-junoscript>
</junoscript>
```

Description

Indicate that the server has encountered a problem while processing the client application's request. The child tag elements described in the Contents section detail the nature of the warning.

Attributes

xmlns—XML namespace for the contents of the tag element. The value is a URL of the form <http://xml.juniper.net/xnm/version/xnm>, where *version* is a string such as "1.1".

xmlns:xnm—521XML namespace for child tag elements that have the **xnm:** prefix in their names. The value is a URL of the form <http://xml.juniper.net/xnm/version/xnm>, where *version* is a string such as "1.1".

- Contents**
- <column>**—(Occurs only during loading of a configuration file) Identifies the element that caused the problem by specifying its position as the number of characters after the first character in the specified line in the configuration file that was being loaded. The line and file are specified by the accompanying **<line-number>** and **<filename>** tag elements.
 - <edit-path>**—(Occurs only during loading of configuration data) Specifies the path to the configuration hierarchy level at which the problem occurred, in the form of the CLI configuration mode banner.
 - <filename>**—(Occurs only during loading of a configuration file) Names the configuration file that was being loaded.
 - <line-number>**—(Occurs only during loading of a configuration file) Specifies the line number where the problem occurred in the configuration file that was being loaded, which is named by the accompanying **<filename>** tag element.
 - <message>**—Describes the warning in a natural-language text string.
 - <source-daemon>**—Names the Junos OS module that was processing the request in which the warning occurred.
 - <statement>**—(Occurs only during loading of configuration data) Identifies the configuration statement that was being processed when the error occurred. The accompanying **<edit-path>** tag element specifies the statement's parent hierarchy level.
 - <token>**—Names which element in the request caused the warning.

The other tag element is explained separately.

- Related Documentation**
- [Handling an Error or Warning in Junos XML Protocol Sessions on page 84](#)
 - [<junoscript> on page 123](#)
 - [<reason> on page 132](#)
 - [<xnm:error> on page 134](#)

Junos XML Element Attributes

active

Usage

```
<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent of the element -->
      <element active="active">
        <name>identifier</name> <!-- if element has identifier -->
      </element>
      <!-- closing tag for each parent of the element -->
    </configuration>
  </load-configuration>
</rpc>
```

Description Reactivate a previously deactivated configuration element.

The **active** attribute can be combined with one or more of the **insert**, **rename**, or **replace** attributes. To deactivate an element, include the **inactive** attribute instead.

- Related Documentation**
- [Changing a Configuration Element's Activation State Using the Junos XML Protocol on page 203](#)
 - [Changing a Configuration Element's Activation State Simultaneously with Other Changes Using the Junos XML Protocol on page 208](#)
 - [inactive on page 141](#)
 - [insert on page 142](#)
 - [<load-configuration> on page 114](#)
 - [rename on page 151](#)
 - [replace on page 152](#)
 - [<rpc> on page 121](#)

count

Usage

```
<rpc>
  <get-configuration>
```

```
<configuration>
  <!-- opening tags for each parent of the object -->
    <object-type count="count"/>
  <!-- closing tags for each parent of the object -->
</configuration>
</get-configuration>
</rpc>
```

Description Specify the number of configuration objects of the specified type about which to return information. If the attribute is omitted, the Junos XML protocol server returns information about all objects of the type.

The attribute can be combined with one or more of the **matching**, **recurse**, and **start** attributes.

If the application requests Junos XML-tagged output (the default), the Junos XML protocol server includes two attributes for each returned object:

- **junos:position**—Specifies the object's numerical index.
- **junos:total**—Reports the total number of such objects that exist in the hierarchy.

These attributes do not appear if the application requests formatted ASCII output by including the **format="text"** attribute in the opening **<get-configuration>** tag.

- Related Documentation**
- [Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol on page 307](#)
 - [<get-configuration> on page 109](#)
 - [matching on page 149](#)
 - [recurse on page 150](#)
 - [<rpc> on page 121](#)
 - [start on page 154](#)

delete

Usage

```
<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent of the element -->

      <!-- For a hierarchy level or object without an identifier -->
        <level-or-object delete="delete">

      <!-- For an object with an identifier (here, called <name>) -->
        <object delete="delete">
          <name>identifier</name>
        </object>

      <!-- For a single-value or fixed-form option of an object -->
        <object>
          <name>identifier</name> <!-- if object has identifier -->
```

```

        <option delete="delete"/>
    </object>

    <!-- closing tag for each parent of the element -->

    <!-- For a value in a multivalued option of an object -->
    <!-- opening tag for each parent of the parent object -->
        <parent-object>
            <name>identifier</name>
            <object delete="delete">value</object>
        </parent-object>
    <!-- closing tag for each parent of the parent object -->

</configuration>
</load-configuration>
</rpc>

```

Description Specify that the Junos XML protocol server remove the configuration element from the candidate configuration or open configuration database. The only acceptable value for the attribute is "delete".

Related Documentation

- [Deleting Elements in Configuration Data Using the Junos XML Protocol on page 186](#)
- [<load-configuration> on page 114](#)
- [<rpc> on page 121](#)

inactive

Usage

```

<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent of the element -->

      <!-- if immediately deactivating a newly created element -->
      <element inactive="inactive">
        <name>identifier</name> <!-- if element has identifier -->
        <!-- tag elements for each child of the element -->
      </element>

      <!-- if deactivating an existing element -->
      <element inactive="inactive">
        <name>identifier</name> <!-- if element has identifier -->
      </element>
      <!-- closing tag for each parent of the element -->
    </configuration>
  </load-configuration>
</rpc>

```

Description Deactivate a configuration element when loading configuration data into the candidate configuration or open configuration database using the **<load-configuration>** operation. When the configuration is committed, the element remains in the configuration, but the element does not affect the functioning of the routing, switching, or security platform.

The **inactive** attribute can be combined with one or more of the **insert**, **rename**, or **replace** attributes, as described in [“Changing a Configuration Element’s Activation State Simultaneously with Other Changes Using the Junos XML Protocol” on page 208](#). To reactivate a deactivated element, include the **active** attribute instead.

- | | |
|------------------------------|--|
| Related Documentation | <ul style="list-style-type: none">• Changing a Configuration Element’s Activation State Using the Junos XML Protocol on page 203• active on page 139• insert on page 142• <load-configuration> on page 114• rename on page 151• <rpc> on page 121 |
|------------------------------|--|

insert

Usage	<pre><rpc> <load-configuration> <configuration> <!-- opening tag for each parent of the set --> <ordered-set insert="first"> <name>identifier-for-moving-object</name> </ordered-set> <!-- if each element in the ordered set has one identifier --> <ordered-set insert="(before after)" name="referent-value"> <name>identifier-for-moving-object</name> </ordered-set> <!-- if each element in the ordered set has two identifiers --> <ordered-set insert="(before after)" identifier1="referent-value" identifier2="referent-value"> <identifier1>value-for-moving-object</identifier1> <identifier2>value-for-moving-object</identifier2> </ordered-set> <!-- closing tag for each parent of the set --> </configuration> </load-configuration> </rpc></pre>
--------------	--

Description	<p>Change the position of an existing configuration element in an ordered set. The insert="first" attribute moves the existing element to the first position in the list. The insert="before" and insert="after" attributes specify the new position relative to a reference element, which is specified by including an attribute named after each of its identifier tags. In the Usage section, the identifier tag element is called <name> when each element in the set has one identifier.</p>
--------------------	--

The **insert** attribute can be combined with either the **active** or **inactive** attribute, as described in [“Changing a Configuration Element’s Activation State Simultaneously with Other Changes Using the Junos XML Protocol”](#) on page 208.

- | | |
|------------------------------|---|
| Related Documentation | <ul style="list-style-type: none"> • Reordering Elements In Configuration Data Using the Junos XML Protocol on page 197 • active on page 139 • inactive on page 141 • <load-configuration> on page 114 • <rpc> on page 121 |
|------------------------------|---|

junos:changed

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration standard-attributes junos:changed="changed">
    <!-- opening-tag-for-each-parent-level junos:changed="changed" -->

    <!-- If the changed element is an empty tag -->
    <element junos:changed="changed"/>

    <!-- If the changed element has child tag elements -->
    <element junos:changed="changed">
      <first-child-of-element junos:changed="changed">
        <second-child-of-element junos:changed="changed">
          <!-- additional children of element - ->
        </element>
      </element>

    <!-- closing-tag-for-each-parent-level -->
  </configuration>
</rpc-reply>
```

Description Indicate that a configuration element has changed since the last commit operation. The Junos XML protocol server includes the attribute when the client application includes the **changed** attribute in a **<get-configuration>** operation. The attribute appears in the opening tag of every parent tag element in the path to the changed configuration element, including the opening top-level **<configuration>** tag.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the empty **<get-configuration/>** tag or opening **<get-configuration>** tag.

For information about the standard attributes in the opening **<configuration>** tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session”](#) on page 272.

- | | |
|------------------------------|---|
| Related Documentation | <ul style="list-style-type: none"> • Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol on page 298 • <get-configuration> on page 109 |
|------------------------------|---|

- [<rpc-reply> on page 134](#)

junos:changed-localtime

Usage	<pre><rpc-reply xmlns:junos="URL"> <configuration junos:changed-seconds="seconds" \ junos:changed-localtime="YYYY-MM-DD hh:mm:ss TZ"> <!-- Junos XML tag elements for the requested configuration data --> </configuration> </rpc-reply></pre>
Description	(Displayed when the candidate configuration is requested) Specify the time when the configuration was last changed as the date and time in the device's local time zone.
Related Documentation	<ul style="list-style-type: none">• Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session on page 272• <configuration> in the Junos XML API Configuration Developer Reference• <rpc-reply> on page 134• junos:changed-seconds on page 144

junos:changed-seconds

Usage	<pre><rpc-reply xmlns:junos="URL"> <configuration junos:changed-seconds="seconds" \ junos:changed-localtime="YYYY-MM-DD hh:mm:ss TZ"> <!-- Junos XML tag elements for the requested configuration data --> </configuration> </rpc-reply></pre>
Description	(Displayed when the candidate configuration is requested) Specify the time when the configuration was last changed as the number of seconds since midnight on 1 January 1970.
Related Documentation	<ul style="list-style-type: none">• Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session on page 272• <configuration> in the Junos XML API Configuration Developer Reference• <rpc-reply> on page 134• junos:changed-localtime on page 144

junos:commit-localtime

Usage	<pre><rpc-reply xmlns:junos="URL"> <configuration junos:commit-seconds="seconds" \ junos:commit-localtime="YYYY-MM-DD hh:mm:ss TZ" \</pre>
-------	---


```

        junos:commit-user="username">
        <!-- Junos XML tag elements for the requested configuration data -->
    </configuration>
</rpc-reply>

```

Description (Displayed when the active configuration is requested) Specify the time when the configuration was committed as the date and time in the device's local time zone.

- Related Documentation**
- [Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session on page 272](#)
 - **<configuration>** in the *Junos XML API Configuration Developer Reference*
 - **<rpc-reply>** on page 134
 - **junos:commit-user** on page 145
 - **junos:commit-seconds** on page 145

junos:commit-seconds

Usage

```

<rpc-reply xmlns:junos="URL">
  <configuration junos:commit-seconds="seconds" \
    junos:commit-localtime="YYYY-MM-DD hh:mm:ss TZ" \
    junos:commit-user="username">
    <!-- Junos XML tag elements for the requested configuration data -->
  </configuration>
</rpc-reply>

```

Description (Displayed when the active configuration is requested) Specify the time when the configuration was committed as the number of seconds since midnight on 1 January 1970.

- Related Documentation**
- [Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session on page 272](#)
 - **<configuration>** in the *Junos XML API Configuration Developer Reference*
 - **<rpc-reply>** on page 134
 - **junos:commit-user** on page 145
 - **junos:commit-localtime** on page 144

junos:commit-user

Usage

```

<rpc-reply xmlns:junos="URL">
  <configuration junos:commit-seconds="seconds" \
    junos:commit-localtime="YYYY-MM-DD hh:mm:ss TZ" \
    junos:commit-user="username">
    <!-- Junos XML tag elements for the requested configuration data -->
  </configuration>
</rpc-reply>

```

Description (Displayed when the active configuration is requested) Specify the Junos OS username of the user who requested the commit operation.

- Related Documentation**
- [Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session on page 272](#)
 - **<configuration>** in the *Junos XML API Configuration Developer Reference*
 - **<rpc-reply>** on page 134
 - **junos:commit-localtime** on page 144
 - **junos:commit-seconds** on page 145

junos:group

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration>
    <!-- opening tag for each parent of the element -->
    <inherited-element junos:group="source-group">
      <inherited-child-of-inherited-element junos:group="source-group">
        <!-- inherited-children-of-child junos:group="source-group" -->
        </inherited-child-of-inherited-element>
      </inherited-element>
    <!-- closing tag for each parent of the element -->
  </configuration>
</rpc-reply>
```

Description Name the configuration group from which each configuration element is inherited. The Junos XML protocol server includes the attribute when the client application includes the **inherit** and **groups** attributes in a **<get-configuration>** operation.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the **<get-configuration>** operation. Instead, the Junos XML protocol server provides the information in a comment directly above each inherited element.

- Related Documentation**
- [Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol on page 283](#)
 - **<get-configuration>** on page 109
 - **<rpc-reply>** on page 134

junos:interface-range

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <interfaces>
      <!-- For each inherited element -->
      <interface junos:interface-range="source-interface-range">
```

```

<inherited-element junos:interface-range="source-interface-range">
  <inherited-child-of-inherited-element
    junos:interface-range="source-interface-range">
      <!-- inherited-children-of-child
        junos:interface-range="source-interface-range" -->
      </inherited-child-of-inherited-element>
    </inherited-element>
  </interface>
</interfaces>
</configuration>
</rpc-reply>

```

Description Name the interface range from which each configuration element is inherited. The Junos XML protocol server includes the attribute when the client application includes the **inherit** and **interface-ranges** attributes in a **<get-configuration>** operation.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the **<get-configuration>** operation.

- Related Documentation**
- [Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol on page 283](#)
 - [<get-configuration> on page 109](#)
 - [<rpc-reply> on page 134](#)

junos:key

Usage

```

<rpc-reply xmlns:junos="URL">
  <configuration>
    <!-- opening tag for each parent of the object -->
    <object>
      <name junos:key="key">identifier</name>
      <!-- additional children of object -->
    </object>
    <!-- closing tag for each parent of the object -->
  </configuration>
</rpc-reply>

```

Description Indicate that a child configuration tag element is the identifier for its parent tag element. The Junos XML protocol server includes the attribute when the client application requests information about an object type (with the **<get-configuration>** tag element) and has included the **junos:key** attribute in either the **<get-configuration>** tag or in the opening **<junoscript>** tag for the current session.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the empty **<get-configuration/>** tag or opening **<get-configuration>** tag.

When requesting configuration data with Perl client applications using the `get_configuration()` method, the Junos XML protocol server includes the **junos:key="key"**

attribute in the configuration data output when the client application sets the value of the **Junos::Device** constructor argument **junos_key** to 1. The default value is 0.

The attribute does not appear if the Perl client application requests formatted ASCII output by including the **format=>'text'** attribute in the options for the **get_configuration()** method.

- | | |
|------------------------------|---|
| Related Documentation | <ul style="list-style-type: none">• Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol on page 295• <get-configuration> on page 109• <junoscript> on page 123• <rpc> on page 121 |
|------------------------------|---|

junos:position

Usage	<pre><rpc-reply xmlns:junos="URL"> <configuration> <!-- opening tags for each parent of the object --> <object junos:position="index" junos:total="total" > <!-- closing tags for each parent of the object --> </configuration> </rpc-reply></pre>
--------------	---

Description	<p>Specify the index number of the configuration object in the list of objects of a specified type about which information is being returned. The Junos XML protocol server includes the attribute when the client application requests information about an object type (with the <get-configuration> tag element) and includes the count attribute, the start attribute, or both, in the opening tag for the object type.</p>
--------------------	--

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the opening **<get-configuration>** tag.

- | | |
|------------------------------|--|
| Related Documentation | <ul style="list-style-type: none">• Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol on page 307• count on page 139• <get-configuration> on page 109• junos:total on page 148• <rpc> on page 121• start on page 154 |
|------------------------------|--|

junos:total

Usage	<pre><rpc-reply xmlns:junos="URL"> <configuration></pre>
--------------	--

```

        <!-- opening tags for each parent of the object -->
        <object junos:position="index" junos:total="total">
        <!-- closing tags for each parent of the object -->
    </configuration>
</rpc-reply>

```

Description Specify the number of configuration objects of a specified type about which information is being returned. The Junos XML protocol server includes the attribute when the client application requests information about an object type (with the `<get-configuration>` tag element) and includes the **count** attribute, the **start** attribute, or both, in the opening tag for the object type.

The attribute does not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the opening `<get-configuration>` tag.

- Related Documentation**
- [Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol on page 307](#)
 - [count on page 139](#)
 - [<get-configuration> on page 109](#)
 - [junos:position on page 148](#)
 - [<rpc> on page 121](#)
 - [start on page 154](#)

matching

Usage

```

<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the level -->
      <level matching="matching-expression"/>
      <!-- closing tags for each parent of the level -->
    </configuration>
  </get-configuration>
</rpc>

```

Description Request information about only those instances of a configuration object type at the specified level in the configuration hierarchy that have the specified set of characters in their identifier names (characters that match a regular expression). If the attribute is omitted, the Junos XML protocol server returns the complete set of child tag elements for the specified parent level.

The attribute can be combined with one or more of the **count**, **recurse**, and **start** attributes.

To represent the objects to return, the *matching-expression* value uses a slash-separated list of hierarchy level and object names similar to an XML Path Language (XPath) representation. Each level in the representation can be either a full level name or a regular expression that matches the identifier name of one or more instances of an object type:

`object-type[name='regular-expression']"`

The regular expression uses the notation defined in POSIX Standard 1003.2 for extended (modern) UNIX regular expressions. For details about the notation, see ["Requesting Subsets of Configuration Objects Using Regular Expressions"](#) on page 314.

- | | |
|------------------------------|---|
| Related Documentation | <ul style="list-style-type: none">• Requesting Subsets of Configuration Objects Using Regular Expressions on page 314• count on page 139• <get-configuration> on page 109• <rpc> on page 121• start on page 154 |
|------------------------------|---|

protect

Usage	<pre><rpc> <load-configuration> <configuration> <!-- opening tag for each parent of the element --> <element protect="protect"> <name>identifier</name> <!-- if element has identifier --> </element> <!-- closing tag for each parent of the element --> </configuration> </load-configuration> </rpc></pre>
--------------	---

Release Information	Attribute introduced in Junos OS Release 11.2.
----------------------------	--

Description	Protect a configuration element from being modified or deleted. The protect attribute can be applied to configuration hierarchies or individual configuration statements. The protect attribute can be combined with the active and inactive attributes. To unprotect a protected element, include the unprotect attribute instead.
--------------------	--

- | | |
|------------------------------|--|
| Related Documentation | <ul style="list-style-type: none">• Protecting or Unprotecting a Configuration Object Using the Junos XML Protocol on page 200• Example: Protecting the Junos OS Configuration from Modification or Deletion• <load-configuration> on page 114• <rpc> on page 121• unprotect on page 154 |
|------------------------------|--|

recurse

Usage	<pre><rpc> <get-configuration> <configuration></pre>
--------------	--

```

        <!-- opening tags for each parent of the object -->
        <object-type recurse="false"/>
        <!-- closing tags for each parent of the object -->
    </configuration>
</get-configuration>
</rpc>

```

Description Request only the identifier tag element for each configuration object of a specified type in the configuration hierarchy. If the attribute is omitted, the Junos XML protocol server returns the complete set of child tag elements for every object. The only acceptable value for the attribute is "false".

The attribute can be combined with one or more of the **count**, **matching**, and **start** attributes.

- Related Documentation**
- [Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol on page 310](#)
 - [count on page 139](#)
 - [<get-configuration> on page 109](#)
 - [<rpc> on page 121](#)
 - [start on page 154](#)

rename

Usage

```

<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent of the object -->

      <!-- if the object has one identifier -->
      <object rename="rename" name="new-name">
        <name>current-name</name>
      </object>

      <!-- if the object has two identifiers, both changing -->
      <object rename="rename" identifier1="new-name" \
        identifier2=new-name">
        <identifier1>current-name</identifier1>
        <identifier2>current-name</identifier2>
      </object>

      <!-- closing tag for each parent of the object -->
    </configuration>
  </load-configuration>
</rpc>

```

Description Change the name of one or more of a configuration object's identifiers. In the Usage section, the identifier tag element is called **<name>** when the element has one identifier.

The **rename** attribute can be combined with either the **inactive** or **active** attribute.

- | | |
|------------------------------|--|
| Related Documentation | <ul style="list-style-type: none">• Renaming Objects In Configuration Data Using the Junos XML Protocol on page 194• active on page 139• inactive on page 141• <load-configuration> on page 114• <rpc> on page 121 |
|------------------------------|--|

replace

Usage	<pre><rpc> <load-configuration action="replace"> <configuration> <!-- opening tag for each parent of the element --> <container-tag replace="replace"> <name>identifier</name> <!-- tag elements for other children, if any --> </container-tag> <!-- closing tag for each parent of the element --> </configuration> </load-configuration> </rpc></pre>
--------------	--

Description	<p>Specify that the configuration element completely replace the element that has the same identifier (in the Usage section, the identifier tag element is called <name>) in the candidate configuration. If the attribute is omitted, the Junos XML protocol server merges the element with the existing element as described in “Merging Elements in Configuration Data Using the Junos XML Protocol” on page 177. The only acceptable value for the attribute is "replace".</p>
--------------------	---

The client application must also include the **action="replace"** attribute in the opening **<load-configuration>** tag.

The **replace** attribute can be combined with either the **active** or **inactive** attribute, as described in [“Changing a Configuration Element’s Activation State Simultaneously with Other Changes Using the Junos XML Protocol” on page 208](#).

- | | |
|------------------------------|--|
| Related Documentation | <ul style="list-style-type: none">• Replacing Elements in Configuration Data Using the Junos XML Protocol on page 181• active on page 139• inactive on page 141• <load-configuration> on page 114• <rpc> on page 121 |
|------------------------------|--|

replace-pattern

Usage

```
<rpc>
  <load-configuration>

    <!-- replace a pattern globally -->
    <configuration replace-pattern="pattern1" with="pattern2" [upto="n"]>
    </configuration>

    <!-- replace a pattern at a specific hierarchy level -->
    <configuration>
      <!-- opening tag for each parent element -->
      <level-or-object replace-pattern="pattern1" with="pattern2"
        [upto="n"]/>
      <!-- closing tag for each parent element -->
    </configuration>

    <!-- replace a pattern for an object that has an identifier -->
    <configuration>
      <!-- opening tag for each parent element -->
      <container-tag replace-pattern="pattern1" with="pattern2"
        [upto="n"]>
        <name>identifier</name>
      </container-tag>
      <!-- closing tag for each parent element -->
    </configuration>

  </load-configuration>
</rpc>
```

Description Replace a variable or identifier in the candidate configuration or open configuration database. Junos OS replaces the pattern specified by the **replace-pattern** attribute with the replacement pattern defined by the **with** attribute. The optional **upto** attribute limits the number of objects replaced. The scope of the replacement is determined by the placement of the attributes in the configuration data.

Attributes **replace-pattern="*pattern1*"**—Text string or regular expression that defines the identifiers or values you want to match.

with="*pattern2*"—Text string or regular expression that replaces the identifiers and values located with *pattern1*.

upto="*n*"—Number of objects replaced. The value of *n* controls the total number of objects that are replaced in the configuration (not the total number of times the pattern occurs). Objects at the same hierarchy level (siblings) are replaced first. Multiple occurrences of a pattern within a given object are considered a single replacement. If you do not include the **upto** attribute or you set the attribute equal to zero, all identifiers and values in the configuration that match the pattern are replaced.

Range: 1 through 4294967295

Default: 0

- Related Documentation**
- [Replacing Patterns in Configuration Data Using the NETCONF or Junos XML Protocol on page 212](#)
 - *Using Global Replace in the Junos OS Configuration*
 - *Common Regular Expressions to Use with the replace Command*
 - *replace*

start

Usage

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object -->
      <object-type start="index"/>
      <!-- closing tags for each parent of the object -->
    </configuration>
  </get-configuration>
</rpc>
```

Description Specify the index number of the first object to return (1 for the first object, 2 for the second, and so on) when requesting information about a configuration object of a specified type. If the attribute is omitted, the returned set of objects starts with the first one in the configuration hierarchy.

The attribute can be combined with one or more of the **count**, **matching**, and **recurse** attributes.

If the application requests Junos XML-tagged output (the default), the Junos XML protocol server includes two attributes for each returned object:

- **junos:position**—Specifies the object's numerical index.
- **junos:total**—Reports the total number of such objects that exist in the hierarchy.

These attributes do not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the opening **<get-configuration>** tag.

- Related Documentation**
- [Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol on page 307](#)
 - [count on page 139](#)
 - [<get-configuration> on page 109](#)
 - [recurse on page 150](#)
 - [<rpc> on page 121](#)

unprotect

Usage

```
<rpc>
```

```

<load-configuration>
  <configuration>
    <!-- opening tag for each parent of the element -->
    <element unprotect="unprotect">
      <name>identifier</name> <!-- if element has identifier -->
    </element>
    <!-- closing tag for each parent of the element -->
  </configuration>
</load-configuration>
</rpc>

```

Release Information Attribute introduced in Junos OS Release 11.2.

Description Unprotect a previously protected configuration element. The **unprotect** attribute cannot be combined with other attributes such as **active**, **inactive**, **rename**, or **replace**. If an element is protected, a request to simultaneously unprotect and modify the element will unprotect the element, but it will also produce a warning message that the additional modification cannot be completed because the element is protected. You must unprotect the element first and then make the modification.

- Related Documentation**
- [Protecting or Unprotecting a Configuration Object Using the Junos XML Protocol on page 200](#)
 - *Example: Protecting the Junos OS Configuration from Modification or Deletion*
 - [<load-configuration> on page 114](#)
 - [<rpc> on page 121](#)
 - [protect on page 150](#)

xmlns

Usage

```

<rpc-reply xmlns:junos="URL">
  <operational-response xmlns="URL-for-DTD">
    <!-- Junos XML tag elements for the requested information -->
  </operational-response>
</rpc-reply>

```

Description Define the XML namespace for the enclosed tag elements that do not have a prefix (such as **junos**;) in their names. The namespace indicates which Junos XML document type definition (DTD) defines the set of tag elements in the response.

- Related Documentation**
- [Requesting Operational Information Using the Junos XML Protocol on page 255](#)
 - [<rpc-reply> on page 134](#)

PART 3

Managing Configurations Using the Junos XML Protocol

- [Changing the Configuration Using the Junos XML Protocol on page 159](#)
- [Committing the Configuration on a Device Using the Junos XML Protocol on page 217](#)
- [Using the Ephemeral Configuration Database on page 237](#)

CHAPTER 11

Changing the Configuration Using the Junos XML Protocol

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Uploading Configuration Data as a File Using the Junos XML Protocol on page 162](#)
- [Uploading Configuration Data as a Data Stream Using the Junos XML Protocol on page 164](#)
- [Defining the Format of Configuration Data to Upload in a Junos XML Protocol Session on page 166](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)
- [Creating New Elements in Configuration Data Using the Junos XML Protocol on page 175](#)
- [Merging Elements in Configuration Data Using the Junos XML Protocol on page 177](#)
- [Replacing Elements in Configuration Data Using the Junos XML Protocol on page 181](#)
- [Replacing Only Updated Elements in Configuration Data Using the Junos XML Protocol on page 184](#)
- [Deleting Elements in Configuration Data Using the Junos XML Protocol on page 186](#)
- [Renaming Objects In Configuration Data Using the Junos XML Protocol on page 194](#)
- [Reordering Elements In Configuration Data Using the Junos XML Protocol on page 197](#)
- [Protecting or Unprotecting a Configuration Object Using the Junos XML Protocol on page 200](#)
- [Changing a Configuration Element's Activation State Using the Junos XML Protocol on page 203](#)
- [Changing a Configuration Element's Activation State Simultaneously with Other Changes Using the Junos XML Protocol on page 208](#)
- [Replacing Patterns in Configuration Data Using the NETCONF or Junos XML Protocol on page 212](#)

Requesting Configuration Changes Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, you can use Junos XML protocol operations along with Junos XML, command-line interface (CLI) configuration statements, **set** commands, or JSON data to change the configuration on a routing, switching, or security platform. The Junos XML protocol **<load-configuration>** operation and its attributes offer functionality that is analogous to configuration mode commands in the Junos OS CLI. The Junos XML tag elements described here correspond to configuration statements, which are described in the Junos OS configuration guides.

To change the configuration on a device running Junos OS, the client application performs the procedures described in the indicated sections:

1. Establishes a connection to the Junos XML protocol server on the routing, switching, or security platform, as described in [“Connecting to the Junos XML Protocol Server” on page 68](#).
2. Starts a Junos XML protocol session, as described in [“Starting Junos XML Protocol Sessions” on page 70](#).
3. Optionally locks the candidate configuration, creates a private copy of the candidate configuration, or opens an instance of the ephemeral configuration database.

Locking the configuration prevents other users or applications from changing it at the same time. Creating a private copy enables the application to make changes without affecting the candidate configuration until the copy is committed. For more information, see [“Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol” on page 86](#).

For information about the ephemeral configuration database, see [“Understanding the Ephemeral Configuration Database” on page 237](#) and [“Enabling and Configuring Instances of the Ephemeral Configuration Database” on page 244](#).

4. Encloses the **<load-configuration>** tag element in an **<rpc>** tag element. By including various attributes in the **<load-configuration>** tag, the application can provide the configuration data either in a file or as a directly loaded tag stream, and as Junos XML tag elements, formatted ASCII text, JSON-formatted data, or Junos OS configuration mode **set** commands. The client application can specify that the configuration data completely replace the existing candidate configuration, or the application can specify the manner in which the Junos XML protocol server loads the data into the existing candidate configuration, private copy, or open instance of the ephemeral configuration database. The basic syntax is as follows:

```
<rpc>
  <!-- If providing configuration data in a file -->
    <load-configuration url="file" [optional attributes] />

  <!-- If providing configuration data in a data stream -->
    <load-configuration [optional attributes] >
      <!-- configuration data -->
    </load-configuration>
</rpc>
```


5. Accepts the tag stream emitted by the Junos XML protocol server in response to each request and extracts its content, as described in [“Parsing the Junos XML Protocol Server Response” on page 80](#).

The Junos XML protocol server confirms that it incorporated the configuration data by returning the `<load-configuration-results>` tag element and `<load-success/>` tag in the `<rpc-reply>` tag element:

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

If the load operation fails, the `<load-configuration-results>` tag element instead encloses the `<load-error-count>` tag element, which indicates the number of errors that occurred. In this case, the application or an administrator must eliminate the errors before committing the configuration.

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-error-count>count</load-error-count>
  </load-configuration-results>
</rpc-reply>
```

6. (Optional) Verifies the syntactic correctness of the candidate configuration or a private copy before attempting to commit it, as described in [“Verifying Configuration Syntax Using the Junos XML Protocol” on page 217](#).
7. Commits changes made to the candidate configuration or private copy, as described in [“Committing the Candidate Configuration Using the Junos XML Protocol” on page 218](#), or commits changes made to an open instance of the ephemeral configuration database, as described in [“Committing an Instance of the Ephemeral Configuration Database Using the NETCONF or Junos XML Protocol” on page 249](#).
8. Unlocks the candidate configuration if it is locked, or closes a private copy of the configuration or an open instance of the ephemeral configuration database.

Other users and applications cannot change the configuration while it remains locked. For more information, see [“Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol” on page 86](#).
9. Ends the Junos XML protocol session and closes the connection to the device, as described in [“Ending a Junos XML Protocol Session and Closing the Connection” on page 90](#).

Related Documentation

- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Uploading and Formatting Configuration Data in a Junos XML Protocol Session

In a Junos XML protocol session with a device running Junos OS, a client application uses the **<load-configuration>** tag element to upload configuration data to the candidate configuration or open configuration database on the device. By setting the **<load-configuration>** attributes to the appropriate values, the client application can specify the source and format of the configuration data to upload. A client application can provide new configuration data using a text file or streaming data, and the data can be formatted as Junos XML tag elements, formatted ASCII text, JSON data, or a series of Junos OS configuration mode commands.

You can choose to stream your configuration changes within your session or reference data files that include the desired configuration changes. Each method has advantages and disadvantages. Streaming data allows you to send your configuration change data inline, using your Junos XML protocol connection. This is useful when the device is behind a firewall and you cannot establish another connection to upload a data file. With text files you can keep the edit configuration commands simple; with data files, there is no need to include the possibly complex configuration data stream.

The delivery mechanism and the format are discussed in detail in the following topics:

- [Uploading Configuration Data as a File Using the Junos XML Protocol on page 162](#)
- [Uploading Configuration Data as a Data Stream Using the Junos XML Protocol on page 164](#)
- [Defining the Format of Configuration Data to Upload in a Junos XML Protocol Session on page 166](#)

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Uploading Configuration Data as a File Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to upload configuration data stored in a file, a client application encloses the **<load-configuration/>** tag with the **url** attribute in an **<rpc>** tag element.

If the data is Junos XML tag elements, either include the **format="xml"** attribute or omit the **format** attribute, which defaults to XML.

```
<rpc>
  <load-configuration url="file-location" />
</rpc>
```

If the data is formatted ASCII text, include the **format="text"** attribute.

```
<rpc>
  <load-configuration url="file-location" format="text"/>
</rpc>
```

If the data is configuration mode **set** commands, include the **action="set"** and **format="text"** attributes.

```
<rpc>
  <load-configuration url="file-location" action="set" format="text"/>
</rpc>
```

Starting in Junos OS Release 16.1, you can load configuration data formatted using JavaScript Object Notation (JSON) on devices running Junos OS. If the data uses JSON format, include the **format="json"** attribute.

```
<rpc>
  <load-configuration url="file-location" format="json"/>
</rpc>
```

Before loading the file, the client application or an administrator saves the configuration data as the contents of the file. Enclose Junos XML tag elements in a **<configuration>** tag element. For information about the syntax for the data in the file, see [“Defining the Format of Configuration Data to Upload in a Junos XML Protocol Session” on page 166](#).



NOTE: Configuration data formatted as ASCII text, Junos OS configuration mode commands, or JSON data is not enclosed in **<configuration-text>**, **<configuration-set>**, or **<configuration-json>** tag elements when it is loaded from a file.

The value of the **url** attribute can be a local file path, an FTP location, or a Hypertext Transfer Protocol (HTTP) URL:

- A local filename can have one of the following forms:
 - **/path/filename**—File on a mounted file system, either on the local flash disk or on hard disk.
 - **a:filename** or **a:path/filename**—File on the local drive. The default path is / (the root-level directory). The removable media can be in MS-DOS or UNIX (UFS) format.

- A filename on an FTP server has the following form:

ftp://username:password@hostname/path/filename

- A filename on an HTTP server has the following form:

http://username:password@hostname/path/filename

In each case, the default value for the *path* variable is the home directory for the username. To specify an absolute path, the application starts the path with the characters **%2F**; for example, **ftp://username:password@hostname/%2Fpath/filename**.

The `url` attribute can be combined with one or more of the following attributes in the `<load-configuration/>` tag:

- `format`
- `action`

The following example shows how to incorporate Junos XML-tagged configuration data stored in the file `/var/configs/user-accounts` on the FTP server called `cfg-server.mycompany.com`. The opening `<load-configuration>` tag appears on two lines for legibility only.

Client Application

```
<rpc>
  <load-configuration \
    url="ftp://admin:AdminPwd@cfg-server.mycompany.com/var/configs/user-accounts"/>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1179

Release History Table

Release	Description
16.1	Starting in Junos OS Release 16.1, you can load configuration data formatted using JavaScript Object Notation (JSON) on devices running Junos OS.

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Uploading Configuration Data as a Data Stream Using the Junos XML Protocol on page 164](#)
- [Defining the Format of Configuration Data to Upload in a Junos XML Protocol Session on page 166](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Uploading Configuration Data as a Data Stream Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to upload configuration data as a data stream, a client application encloses the `<load-configuration>` tag element in an `<rpc>` tag element.

To define the configuration elements to change as Junos XML tag elements, the application emits the tag elements representing all levels of the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to each element to change.

```
<rpc>
  <load-configuration>
    <configuration>
      <!-- tag elements representing the configuration data -->
    </configuration>
  </load-configuration>
</rpc>
```

To define the configuration data to change as formatted ASCII text, the application encloses the statements in a `<configuration-text>` tag element and includes the `format="text"` attribute in the opening `<load-configuration>` tag.

```
<rpc>
  <load-configuration format="text">
    <configuration-text>
      /* formatted ASCII configuration data */
    </configuration-text>
  </load-configuration>
</rpc>
```

To define the configuration data to change as Junos OS configuration mode `set` commands, the application encloses the commands in a `<configuration-set>` tag element and includes the `action="set"` and `format="text"` attributes in the opening `<load-configuration>` tag.

```
<rpc>
  <load-configuration action="set" format="text">
    <configuration-set>
      /* configuration mode commands */
    </configuration-set>
  </load-configuration>
</rpc>
```

Starting in Junos OS Release 16.1, you can load configuration data formatted using JavaScript Object Notation (JSON) on devices running Junos OS. To define the configuration data to change as JSON format, the application encloses the data in a `<configuration-json>` tag element and includes the `format="json"` attribute in the opening `<load-configuration>` tag.

```
<rpc>
  <load-configuration format="json">
    <configuration-json>
      /* JSON-formatted configuration data */
    </configuration-json>
  </load-configuration>
</rpc>
```

For information about the syntax for Junos XML tag elements, formatted ASCII text, configuration mode commands, and JSON format, see [“Defining the Format of Configuration Data to Upload in a Junos XML Protocol Session”](#) on page 166.

Release History Table

Release	Description
16.1	Starting in Junos OS Release 16.1, you can load configuration data formatted using JavaScript Object Notation (JSON) on devices running Junos OS.

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Uploading Configuration Data as a File Using the Junos XML Protocol on page 162](#)
- [Defining the Format of Configuration Data to Upload in a Junos XML Protocol Session on page 166](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Defining the Format of Configuration Data to Upload in a Junos XML Protocol Session

In a Junos XML protocol session with a device running Junos OS, a client application can upload configuration data to the device either in a file or as a data stream emitted during the Junos XML protocol session. In both cases, the client application can use Junos XML tag elements, formatted ASCII text, Junos OS configuration mode commands, or JavaScript Object Notation (JSON) to define the new configuration data.

If the application uses Junos XML tag elements, it includes the tag elements representing all levels of the configuration hierarchy from the root (the **<configuration>** tag element) down to each new or changed element. The notation is the same as that used to request configuration information, and is described in detail in [“Requesting Configuration Changes Using the Junos XML Protocol” on page 160](#).

```
<configuration>
  <!-- tag elements representing the configuration data -->
</configuration>
```

If the application provides the configuration data as formatted ASCII text, it uses the standard Junos OS CLI notation to indicate the hierarchical relationships between configuration statements—the newline character, tabs and other white space, braces, and square brackets. For each new or changed element, the complete statement path is specified, starting with the top-level statement that appears directly under the **[edit]** hierarchy level.

When ASCII text is provided as a data stream, it is enclosed in the **<configuration-text>** tag element.

```
<configuration-text>
  /* formatted ASCII configuration statements */
</configuration-text>
```

When ASCII text is provided in a file, the **<configuration-text>** tag element is not included in the file.

When providing configuration data as ASCII text, the application must also include the **format="text"** attribute in the **<load-configuration>** tag.

```
<rpc>
  <load-configuration url="file-location" format="text"/>
</rpc>

<rpc>
  <load-configuration format="text">
    <configuration-text>
      /* formatted ASCII configuration data */
    </configuration-text>
  </load-configuration>
</rpc>
```

Starting in Junos OS Release 11.4, you can load configuration data as configuration mode **set** commands. When you provide configuration data as configuration mode **set** commands, Junos OS executes the configuration instructions line by line. For each element, you can specify the complete statement path in the command, or you can use navigation commands, such as **edit** and **up**, to move around the configuration hierarchy as you would in CLI configuration mode.

When configuration mode **set** commands are provided as a data stream, the commands are enclosed in the **<configuration-set>** tag element.

```
<configuration-set>
  /* configuration mode commands */
</configuration-set>
```

When configuration mode **set** commands are provided in a file, the **<configuration-set>** tag element is not included in the file.

When providing configuration data as Junos OS configuration mode commands, the application must also include the **action="set"** and **format="text"** attributes in the **<load-configuration>** tag.

```
<rpc>
  <load-configuration url="file-location" action="set" format="text"/>
</rpc>

<rpc>
  <load-configuration action="set" format="text">
    <configuration-set>
      /* configuration mode commands to load */
    </configuration-set>
  </load-configuration>
</rpc>
```

Starting in Junos OS Release 16.1, you can load configuration data formatted using JavaScript Object Notation (JSON) on the device. If the application provides configuration data in JSON format, it includes the configuration data representing all levels of the configuration hierarchy from the root down to each new or changed element.

When configuration data in JSON format is provided as a data stream, the data is enclosed in the `<configuration-json>` tag element.

```
<configuration-json>
  /* JSON-formatted configuration data */
</configuration-json>
```

When configuration data in JSON format is provided in a file, the `<configuration-json>` tag element is not included in the file.

When providing configuration data in JSON format, the application must also include the `format="json"` attribute in the `<load-configuration>` tag.

```
<rpc>
  <load-configuration url="file-location" format="json"/>
</rpc>

<rpc>
  <load-configuration format="json">
    <configuration-json>
      /* JSON-formatted configuration data */
    </configuration-json>
  </load-configuration>
</rpc>
```

The `format` attribute can be combined with one or more of the following attributes:

- `url`
- `action`



NOTE: JSON format is only supported for action values of merge, override, and update.

For reference pages for the `<configuration>`, `<configuration-text>`, `<configuration-set>`, and `<configuration-json>` tag elements, see the *Junos XML API Operational Developer Reference*.

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Uploading Configuration Data as a File Using the Junos XML Protocol on page 162](#)
- [Uploading Configuration Data as a Data Stream Using the Junos XML Protocol on page 164](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session

In a Junos XML protocol session with a device running Junos OS, a client application can upload and replace the entire configuration or specific portions of the configuration by emitting the `<load-configuration>` tag element and including the appropriate child tag elements.

For information about modifying different scopes of configuration information, see the following topics:

- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)
- [Creating New Elements in Configuration Data Using the Junos XML Protocol on page 175](#)
- [Merging Elements in Configuration Data Using the Junos XML Protocol on page 177](#)
- [Replacing Elements in Configuration Data Using the Junos XML Protocol on page 181](#)
- [Replacing Only Updated Elements in Configuration Data Using the Junos XML Protocol on page 184](#)
- [Deleting Elements in Configuration Data Using the Junos XML Protocol on page 186](#)
- [Replacing Patterns in Configuration Data Using the NETCONF or Junos XML Protocol on page 212](#)

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)

Replacing the Candidate Configuration Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, a client application can replace the entire candidate configuration or a private copy of it, either with new data or by rolling back to a previous configuration or a rescue configuration. For instructions about modifying individual configuration elements, see [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#).



NOTE: Junos OS does not support replacing or rolling back the configuration data committed to an instance of the ephemeral configuration database by using either the `action="override"` or `rollback` attribute in the Junos XML protocol `<load-configuration>` operation or by using the `<rollback-config>` RPC.

The following sections discuss how to replace the candidate configuration. The client application must commit the candidate configuration after replacing the data to make it the active configuration on the device.

- [Replacing the Candidate Configuration with New Data on page 170](#)
- [Rolling Back to a Previously Committed Configuration on page 171](#)
- [Replacing the Candidate Configuration with a Rescue Configuration on page 172](#)

Replacing the Candidate Configuration with New Data

To discard the entire candidate configuration or private copy and replace it with new configuration data, a client application includes the **action="override"** attribute in the **<load-configuration>** tag.

```
<rpc>
  <!-- For a file -->
  <load-configuration action="override" url="file" [format="format"]/>

  <!-- For a data stream -->
  <load-configuration action="override" [format="format"]>
    <!-- configuration data -->
  </load-configuration>
</rpc>
```

For more information about the **url** and **format** attributes and the syntax for the new configuration data, see [“Uploading and Formatting Configuration Data in a Junos XML Protocol Session” on page 162](#).

The following example shows how to specify that the contents of the file **/tmp/new.conf** replace the entire candidate configuration. The file contains Junos XML tag elements (the default), so the **format** attribute is omitted.

Client Application

```
<rpc>
  <load-configuration action="override" url="/tmp/new.conf"/>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

Rolling Back to a Previously Committed Configuration

Devices running Junos OS store a copy of the most recently committed configuration and up to 49 previous configurations, depending on the platform. You can roll back to any of the stored configurations. This is useful when configuration changes cause undesirable results, and you want to revert back to a known working configuration. Rolling back the configuration is similar to the process for making configuration changes on the device, but instead of loading configuration data, you perform a rollback, which replaces the entire candidate configuration with a previously committed configuration.

To replace either the candidate configuration or all data in the open configuration database with a previously committed configuration, a Junos XML protocol application can either execute the `<load-configuration/>` operation with the `rollback="index"` attribute, or starting in Junos OS Release 17.4R2, an application can execute the `<rollback-config>` RPC. The `<rollback-config>` RPC is useful when an application does not support executing RPCs that use XML attributes.

To use the `<load-configuration>` operation to replace either the candidate configuration or all data in the open configuration database with a previously committed configuration, a client application includes the `rollback="index"` attribute in the `<load-configuration/>` tag, where *index* is the numerical index of the appropriate previous configuration. Valid values are 0 (zero, for the most recently committed configuration) through one less than the number of stored previous configurations (maximum is 49).

```
<rpc>
  <load-configuration rollback="index">
</rpc>
```

The Junos XML protocol server indicates that the load operation was successful by returning the `<load-configuration-results>` and `<load-success/>` elements in its RPC reply.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.1R1/junos">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

To use the `<rollback-config>` RPC to load a previously committed configuration, a client application emits the `<rollback-config>` element and the `<index>` child element.

```
<rpc>
  <rollback-config>
    <index>1</index>
  </rollback-config>
</rpc>
```

The Junos XML protocol server indicates that the load operation was successful by returning the `<rollback-config-results>` and `<load-success/>` elements in its RPC reply.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.1R1/junos">
  <rollback-config-results>
    <load-success/>
  </rollback-config-results>
</rpc-reply>
```

If the load operation is successful, the client application must commit the configuration to make it the active configuration on the device. If the server encounters an error while loading the rollback configuration, it returns an `<xnm:error>` element with information about the error.

Replacing the Candidate Configuration with a Rescue Configuration

A rescue configuration allows you to define a known working configuration or a configuration with a known state that you can restore at any time. You use the rescue configuration when you need to revert to a known configuration or as a last resort if the device configuration and the backup configuration files become damaged beyond repair. When you create a rescue configuration, the device saves the most recently committed configuration as the rescue configuration.

To replace either the candidate configuration or all data in the open configuration database with the device's rescue configuration, a Junos XML protocol application can either execute the `<load-configuration/>` operation with the `rescue="rescue"` attribute, or starting in Junos OS Release 17.4R2, an application can execute the `<rollback-config>` RPC. The `<rollback-config>` RPC is useful when an application does not support executing RPCs that use XML attributes. The rescue configuration must exist on the device before you can load it.

To use the `<load-configuration/>` operation to replace the candidate configuration with the rescue configuration, include the `rescue="rescue"` attribute in the `<load-configuration/>` tag.

```
<rpc>
  <load-configuration rescue="rescue"/>
</rpc>
```

The Junos XML protocol server indicates that the load operation was successful by returning the `<load-configuration-results>` and `<load-success/>` elements in its RPC reply.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.1R1/junos">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

To use the `<rollback-config>` RPC to load the rescue configuration, a client application emits the `<rollback-config>` element and the `<rescue/>` child tag.

```
<rpc>
  <rollback-config>
    <rescue/>
  </rollback-config>
</rpc>
```

The Junos XML protocol server indicates that the load operation was successful by returning the `<rollback-config-results>` and `<load-success/>` elements in its RPC reply.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.1R1/junos">
  <rollback-config-results>
    <load-success/>
  </rollback-config-results>
</rpc-reply>
```

```
</rollback-config-results>
</rpc-reply>
```

If the load operation is successful, the client application must commit the configuration to make it the active configuration on the device. If the rescue configuration does not exist or the server encounters another error while loading the configuration data, it returns an `<xnm:error>` element with information about the error.

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)

Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, in addition to replacing the entire configuration, a client application can create, modify, or delete one or more configuration elements (hierarchy levels and configuration objects) in the candidate configuration or open configuration database.

To use Junos XML tag elements to represent an element, the application includes the tag elements representing all levels in the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to the element's container tag element. The attributes and child tag elements that are included depend on the operation being performed on the element. The syntax applies both to the contents of a file and to a data stream. In the following example, the identifier tag element is called `<name>`:

```
<configuration>
  <!-- opening tag for each parent of the element -->
  <container-tag [operation-attribute="value"]>
    <name>identifier</name> <!-- if the element has an identifier -->
    <!-- child tag elements --> <!-- if appropriate -->
  </container-tag>
  <!-- closing tag for each parent of the element -->
</configuration>
```

To use formatted ASCII text to represent an element, the application includes the complete statement path, starting with a statement that can appear directly under the `[edit]` hierarchy level. The attributes and child statements to include depend on the operation being performed on the element. The application encloses the set of statements in a `<configuration-text>` tag element when it uploads the configuration data as a data stream. The application omits the `<configuration-text>` tag element when the configuration data is stored in and loaded from a file.

```
<configuration-text>
  /* statements for parent levels of the element */
  operation-to-perform:      # if appropriate
  element identifier {       # if the element has an identifier
    /* child statements */   # if appropriate for the operation
  }
  /* closing braces for parent levels of the element */
</configuration-text>
```

When loading formatted ASCII text, the application must include the **format="text"** attribute in the **<load-configuration>** tag.

To use configuration mode commands to create, modify, or delete an element, the application includes the commands as they would be typed in configuration mode in the CLI. The configuration instructions are executed in the order provided. You can specify the complete statement path in the command, or you can use CLI navigation commands such as **edit** and **up**, to move around the configuration hierarchy.

The application encloses the set of commands in a **<configuration-set>** tag element when it uploads the configuration data as a data stream. The application omits the **<configuration-set>** tag element when the configuration data is stored in and loaded from a file.

```
<configuration-set>
  /* configuration mode commands */
</configuration-set>
```

When loading configuration mode **set** commands, the application must include the **action="set"** and **format="text"** attributes in the **<load-configuration>** tag.

Starting in Junos OS Release 16.1, you can load configuration data formatted using JavaScript Object Notation (JSON) on the device. To use JSON format to represent an element, the application includes JSON objects representing all levels in the configuration hierarchy from the root down to the JSON object representing that element. The attributes and child objects to include depend on the operation being performed on the element. If the attribute value is a Boolean data type, the value is not enclosed in quotes.

The application encloses the JSON data in a **<configuration-json>** tag element when it uploads the configuration data as a data stream. The application omits the **<configuration-json>** tag element when the configuration data is stored in and loaded from a file.

```
<configuration-json>
{
  "configuration" : {
    /* JSON objects for parent levels of the element */
    "container-tag" : {
      "@" : {
        "operation-attribute" : ( "value" | boolean )
      },
      "object" : [
        {
          "@" : {
            "operation-attribute" : ( "value" | boolean )
          },
          "(name | element-identifier)" : "identifier",
          "statement-name" : "statement-value",
          "@statement-name" : {
            "operation-attribute" : ( "value" | boolean )
          },
          /* additional JSON data and child objects */
        }
      ]
    }
  }
  /* closing braces for parent levels of the element */
}
</configuration-json>
```

When loading data in JSON format, the application must include the **format="json"** attribute in the **<load-configuration>** tag.

For more information about the source and formatting for configuration elements, see [“Uploading and Formatting Configuration Data in a Junos XML Protocol Session”](#) on page 162.

For information about the operations a client application can perform on configuration elements, see the following sections:

- [Creating New Elements in Configuration Data Using the Junos XML Protocol](#) on page 175
- [Merging Elements in Configuration Data Using the Junos XML Protocol](#) on page 177
- [Replacing Elements in Configuration Data Using the Junos XML Protocol](#) on page 181
- [Replacing Only Updated Elements in Configuration Data Using the Junos XML Protocol](#) on page 184
- [Deleting Elements in Configuration Data Using the Junos XML Protocol](#) on page 186
- [Renaming Objects In Configuration Data Using the Junos XML Protocol](#) on page 194
- [Reordering Elements In Configuration Data Using the Junos XML Protocol](#) on page 197
- [Protecting or Unprotecting a Configuration Object Using the Junos XML Protocol](#) on page 200
- [Changing a Configuration Element's Activation State Using the Junos XML Protocol](#) on page 203
- [Replacing Patterns in Configuration Data Using the NETCONF or Junos XML Protocol](#) on page 212

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol](#) on page 160
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session](#) on page 162
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session](#) on page 169
- [Replacing the Candidate Configuration Using the Junos XML Protocol](#) on page 169

Creating New Elements in Configuration Data Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to create new configuration elements (hierarchy levels or configuration objects), a client application includes the basic tag elements, formatted ASCII statements, configuration mode commands, or JSON objects described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol”](#) on page 173.

For Junos XML tag elements and formatted ASCII text, new elements can be created in either merge mode or replace mode, which are described in [“Merging Elements in Configuration Data Using the Junos XML Protocol”](#) on page 177 and [“Replacing Elements](#)

in [Configuration Data Using the Junos XML Protocol](#) on page 181. In replace mode, the application includes the **action="replace"** attribute in the **<load-configuration/>** tag or opening **<load-configuration>** tag.

To use Junos XML tag elements to represent the element, the application includes each of the element's identifier tag elements (if it has them) and all child tag elements being defined for the element. In the following, the identifier tag element is called **<name>**. The application does not need to include any attributes in the opening container tag for the new element:

```
<configuration>
  <!-- opening tag for each parent of the element -->
    <container-tag>
      <name>identifier</name>
      <!-- tag elements for other children, if any -->
    </container-tag>
  <!-- closing tag for each parent of the element -->
</configuration>
```

To use formatted ASCII text to represent the element, the application includes each of the element's identifiers (if it has them) and all child statements (with values if appropriate) that it is defining for the element. It does not need to include an operator before the new element:

```
<configuration-text>
  /* statements for parent levels of the element */
  element identifier {
    /* child statements if any */
  }
  /* closing braces for parent levels of the element */
</configuration-text>
```

To use configuration mode commands to create new elements, the application includes the **action="set"** and **format="text"** attributes in the **<load-configuration>** tag. The application includes the **set** command as it would be executed in the CLI. The command includes the statement path to the element, the element's identifier if it has one, and all child statements (with values if appropriate) that it is defining for the element.

```
<configuration-set>
  set statement-path-to-element element identifier child-elements
</configuration-set>
```

Starting in Junos OS Release 16.1, you can load configuration data formatted using JavaScript Object Notation (JSON) on the device. When loading configuration data in JSON format, you can create new elements in merge mode only. To represent the element in JSON, the application includes each element's identifier (if it has one) and all data and child objects being defined for the element. The application does not need to include any specific operation attributes in order to create the new element. In the following example, the JSON member that specifies the element's identifier has the field name "name":

```
<configuration-json>
{
  "configuration" : {
    /* JSON objects for parent levels of the element */
    "container-tag" : {
      "object" : [
        {
          "name" : "identifier",
          /* data and child objects */ # if any
        }
      ]
    }
  }
}
```



```

    }
  ],
  /* data and child objects */ # if any
}
/* closing braces for parent levels of the element */
}
}
</configuration-json>

```

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Merging Elements in Configuration Data Using the Junos XML Protocol

By default, the Junos XML protocol server *merges* loaded configuration data into the candidate configuration according to the following rules. (The rules also apply to a private copy of the configuration or an open instance of the ephemeral configuration database, but for simplicity the following discussion refers to the candidate configuration only.)

- A configuration element (hierarchy level or configuration object) that exists in the candidate but not in the loaded configuration remains unchanged.
- A configuration element that exists in the loaded configuration but not in the candidate is added to the candidate.
- If a configuration element exists in both configurations, the semantics are as follows:
 - If a child statement of the configuration element (represented by a child tag element) exists in the candidate but not in the loaded configuration, it remains unchanged.
 - If a child statement exists in the loaded configuration but not in the candidate, it is added to the candidate.
 - If a child statement exists in both configurations, the value in the loaded configuration replaces the value in the candidate.

Merge mode is the default mode for new configuration elements, so the application simply emits the `<load-configuration>` tag element in an `<rpc>` tag element:

```

<rpc>
  <!-- For a file -->
  <load-configuration url="file" [format="format"]/>

  <!-- For a data stream -->
  <load-configuration [format="format"]>
    <!-- configuration data -->

```

```

        </load-configuration>
    </rpc>

```

For more information about the `url` and `format` attributes, see [“Uploading and Formatting Configuration Data in a Junos XML Protocol Session” on page 162.](#)

To explicitly specify merge mode for configuration data that uses Junos XML elements, formatted ASCII text, or JSON format, the application can include the `action="merge"` attribute in the `<load-configuration>` tag, as shown in the examples at the end of this section.

When using Junos XML tag elements to represent the element to merge into the configuration, the application includes the basic tag elements described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173.](#) It does not include any attributes in the element’s container tag. If adding or changing the value of a child element, the application includes the tag elements for it. If a child remains unchanged, it does not need to be included in the loaded configuration. In the following, the identifier tag element is called `<name>`:

```

<configuration>
  <!-- opening tag for each parent of the element -->
    <container-tag>
      <name>identifier</name> <!-- if the element has an identifier -->
      <!-- tag elements for other children, if any -->
    </container-tag>
  <!-- closing tag for each parent of the element -->
</configuration>

```

When using formatted ASCII text, the application includes the statement path described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173.](#) It does not include a preceding operator, but does include the element’s identifier if it has one. If adding or changing the value of a child element, the application includes the tag elements for it. If a child remains unchanged, it does not need to be included in the loaded configuration.

```

<configuration-text>
/* statements for parent levels of the element */
  element identifier {
    /* child statements if any */
  }
/* closing braces for parent levels of the element */
</configuration-text>

```

Starting in Junos OS Release 16.1, you can load configuration data formatted using JavaScript Object Notation (JSON) on the device. When using JSON to represent the elements to merge into the configuration, the application includes the basic JSON data described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173.](#) It does not need to include any specific operation attributes in the JSON configuration data in order to merge the new or changed element. If adding or changing the value of a child element, the application includes the JSON data or child objects for it. If a child remains unchanged, it does not need to be included in the loaded configuration. In the following example, the JSON member that specifies the element’s identifier has the field name `"name"`:

```

<configuration-json>
{
  "configuration" : {
    /* JSON objects for parent levels of the element */
    "container-tag" : {

```

```

        "object" : [
        {
            "name" : "identifier",
            "statement-name" : "statement-value",    # if any
            /* additional data and child objects */  # if any
        }
        ],
        /* data and child objects */    # if any
    }
    /* closing braces for parent levels of the element */
}
}
</configuration-json>

```

When using configuration mode commands to merge new elements, the application includes the **action="set"** and **format="text"** attributes in the **<load-configuration>** tag, as shown in the examples at the end of this section. The application includes the **set** command, the statement path to the element, and the element's identifier if it has one. If adding or changing the value of a child element, the application includes the child elements or statements in the command. If a child remains unchanged, it does not need to be included.

```

<configuration-set>
    set statement-path-to-element element identifier child-statements
</configuration-set>

```

The following example shows how to merge in a new interface called so-3/0/0 at the **[edit interfaces]** hierarchy level in the configuration. The information is provided as Junos XML tag elements (the default).

Client Application

```

<rpc>
  <load-configuration action="merge">
    <configuration>
      <interfaces>
        <interface>
          <name>so-3/0/0</name>
          <unit>
            <family>
              <inet>
                <address>
                  <name>10.0.0.1/8</name>
                </address>
              </inet>
            </family>
          </unit>
        </interface>
      </interfaces>
    </configuration>
  </load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1131

The following example shows how to use formatted ASCII text to define the same new interface.

Client Application

Junos XML Protocol Server

```

<rpc>
  <load-configuration action="merge" format="text">
    <configuration-text>
      interfaces {
        so-3/0/0 {
          unit 0 {
            family inet {
              address 10.0.0.1/8;
            }
          }
        }
      }
    </configuration-text>
  </load-configuration>
</rpc>

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1132

The following example shows how to use JSON configuration data to define the same interface.

```

<rpc>
<load-configuration format="json">
<configuration-json>
{
  "configuration" : {
    "interfaces" : {
      "interface" : [
        {
          "name" : "so-3/0/0",
          "unit" : [
            {
              "name" : 0,
              "family" : {
                "inet" : {
                  "address" : [
                    {
                      "name" : "10.0.0.1/8"
                    }
                  ]
                }
              }
            }
          ]
        }
      ]
    }
  }
}
</configuration-json>

```

```

</load-configuration>
</rpc>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/16.2R2/junos">
<load-configuration-results>
<load-success/>
</load-configuration-results>
</rpc-reply>

```

The following example shows how to use configuration mode commands to define the same interface.

Client Application

Junos XML Protocol Server

```

<rpc>
  <load-configuration action="set" format="text">
    <configuration-set>
      set interfaces so-3/0/0 unit 0 family inet address 10.0.0.1/8
    </configuration-set>
  </load-configuration>
</rpc>

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1148

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Replacing Elements in Configuration Data Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to replace individual configuration elements (hierarchy levels or configuration objects), a client application emits the `<load-configuration>` tag element with the `action="replace"` attribute in an `<rpc>` tag element.

```

<rpc>
  <!-- For a file -->
  <load-configuration action="replace" url="file" [format="text"]/>

  <!-- For a data stream -->
  <load-configuration action="replace" [format="text"]>
    <!-- configuration data -->
  </load-configuration>
</rpc>

```



NOTE: The `action="replace"` attribute is not supported when loading configuration data into the ephemeral configuration database.

For more information about the `url` and `format` attributes, see [“Uploading and Formatting Configuration Data in a Junos XML Protocol Session” on page 162](#).

To use Junos XML tag elements to define the replacement, the application includes the basic tag elements described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#). Within the container tag, it includes the same child tag elements as for a new element: each of the replacement’s identifier tag elements (if it has them) and all child tag elements being defined for the replacement element. In the following, the identifier tag element is called `<name>`. The application also includes the `replace="replace"` attribute in the opening container tag:

```
<configuration>
  <!-- opening tag for each parent of the element -->
    <container-tag replace="replace">
      <name>identifier</name>
      <!-- tag elements for other children, if any -->
    </container-tag>
  <!-- closing tag for each parent of the element -->
</configuration>
```

To use formatted ASCII text to represent the element, the application includes the complete statement path described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#). As for a new element, it includes each of the replacement’s identifiers (if it has them) and all child statements (with values if appropriate) that it is defining for the replacement. It places the `replace:` statement above the element’s container statement.

```
<configuration-text>
  /* statements for parent levels of the element */
  replace:
  element identifier {
    /* child statements if any */
  }
  /* closing braces for parent levels of the element */
</configuration-text>
```



NOTE: Junos OS does not support using the replace operation when loading JSON-formatted configuration data. To replace configuration elements when using JSON, you must delete the existing element and then add the replacement element.

The following example shows how to grant new permissions for the object named `operator` at the `[edit system login class]` hierarchy level. The information is provided in Junos XML-tagged format (the default).

Client Application

```

<rpc>
  <load-configuration action="replace">
    <configuration>
      <system>
        <login>
          <class replace="replace">
            <name>operator</name>
            <permissions>configure</permissions>
            <permissions>admin-control</permissions>
          </class>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1135

The following example shows how to use formatted ASCII text to make the same change.

Client Application

```

<rpc>
  <load-configuration action="replace" format="text">
    <configuration-text>
      system {
        login {
          replace:
            class operator {
              permissions [ configure admin-control ];
            }
        }
      }
    </configuration-text>
  </load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1136

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)

- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Replacing Only Updated Elements in Configuration Data Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to replace configuration elements (hierarchy levels and configuration objects) only if they differ between the loaded configuration and the candidate configuration or private copy, the application emits the `<load-configuration>` tag element with the `action="update"` attribute in an `<rpc>` tag element.

```
<rpc>
  <!-- For a file -->
    <load-configuration action="update" url="file" [format="format"]/>

  <!-- For a data stream -->
    <load-configuration action="update" [format="format"]>
      <!-- configuration data -->
    </load-configuration>
</rpc>
```

For more information about the `url` and `format` attributes, see ["Uploading and Formatting Configuration Data in a Junos XML Protocol Session" on page 162](#).



NOTE: The `action="update"` attribute is not supported when loading configuration data into the ephemeral configuration database.

This operation is equivalent to the Junos OS CLI `load update` configuration mode command. The Junos OS configuration management software compares the two complete configurations. Each configuration element that is different in the loaded configuration replaces its corresponding element in the existing configuration. Elements that are the same in both configurations remain unchanged. When the configuration is later committed, only system processes that are affected by the changed configuration elements parse the new configuration.

To represent the replacement elements, the application uses the same syntax as for new elements, as described in ["Creating New Elements in Configuration Data Using the Junos XML Protocol" on page 175](#). In the following Junos XML and JSON representations of the configuration, the object identifier is called **name**.

Junos XML elements:

```
<configuration>
  <!-- opening tag for each parent of the element -->
    <container-tag>
      <name>identifier</name>
      <!-- tag elements for other children, if any -->
    </container-tag>
  <!-- closing tag for each parent of the element -->
</configuration>
```


ASCII text:

```
<configuration-text>
  /* statements for parent levels of the element */
  element identifier {
    /* child statements if any */
  }
  /* closing braces for parent levels of the element */
</configuration-text>
```

JSON:

```
<configuration-json>
{
  "configuration" : {
    /* JSON objects for parent levels of the element */
    "container-tag" : {
      "object" : [
        {
          "name" : "identifier",
          "statement-name" : "statement-value", # if any
          /* additional data and child objects */ # if any
        }
      ],
      /* data and child objects */ # if any
    }
    /* closing braces for parent levels of the element */
  }
}
</configuration-json>
```



NOTE: You can load configuration data formatted using JavaScript Object Notation (JSON) starting in Junos OS Release 16.1.

The following example shows how to update the candidate configuration with the contents of the file `/tmp/new.conf` (which resides on the device). The file contains a complete configuration represented as Junos XML tag elements (the default), so the **format** attribute is omitted.

Client Application

```
<rpc>
  <load-configuration action="update" url="/tmp/new.conf"/>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1134

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)

- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Deleting Elements in Configuration Data Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to delete configuration elements (hierarchy levels or configuration objects) from the candidate configuration or open configuration database, a client application emits the basic tag elements described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#). When using Junos XML tag elements to represent the elements to delete, the client application includes the **delete="delete"** attribute in the opening tag for each element. When using formatted ASCII text, the client application precedes each element to delete with the **delete:** operator. When using configuration mode commands to delete elements, the client application uses the **delete** command and specifies the path to the element. When using JSON data to delete an element, the client application includes the **"operation": "delete"** attribute in the attribute list for that element. The placement of the attribute or operator depends on the type of element being deleted, as described in the following sections:



NOTE: You can load configuration data formatted using JavaScript Object Notation (JSON) starting in Junos OS Release 16.1.

- [Deleting a Hierarchy Level or Container Object on page 186](#)
- [Deleting a Configuration Object That Has an Identifier on page 188](#)
- [Deleting a Single-Value or Fixed-Form Option from a Configuration Object on page 190](#)
- [Deleting Values from a Multivalue Option of a Configuration Object on page 192](#)

Deleting a Hierarchy Level or Container Object

To delete a hierarchy level and all of its children (or a container object that has children but no identifier), a client application includes the basic tag elements or configuration statements for its parent levels, as described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#).

When using Junos XML tag elements, the application includes the **delete="delete"** attribute in the empty tag that represents the level or container object:

```
<configuration>
  <!-- opening tag for each parent level -->
    <level-or-object delete="delete"/>
  <!-- closing tag for each parent level -->
</configuration>
```

When using formatted ASCII text, the application places the **delete:** statement above the level to be removed, which is followed by a semicolon (even though in the existing configuration it is followed by curly braces that enclose its child statements):

```
<configuration-text>
```

```

/* statements for parent levels */
delete:
object-or-level;
/* closing braces for parent levels */
</configuration-text>

```

When using configuration mode commands, the application specifies the **delete** command and the statement path to the hierarchy level or object to be removed.

```

<configuration-set>
delete statement-path-to-level-or-object
</configuration-set>

```

When using JSON configuration data to delete a hierarchy level or container object, the application includes the **"operation": "delete"** attribute in the attribute list for the hierarchy or container object:

```

<configuration-json>
{
  "configuration" : {
    /* JSON objects for parent levels */
    "object-or-level" : {
      "@" : {
        "operation" : "delete"
      }
    }
    /* closing braces for parent levels */
  }
}
</configuration-json>

```

The following example shows how to remove the **[edit protocols ospf]** hierarchy level from the candidate configuration using Junos XML tag elements:

Client Application

```

<rpc>
<load-configuration>
<configuration>
<protocols>
<ospf delete="delete"/>
</protocols>
</configuration>
</load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
<load-configuration-results>
<load-success/>
</load-configuration-results>
</rpc-reply>

```

T1137

The following example shows how to remove the **[edit protocols ospf]** hierarchy level from the candidate configuration using configuration data formatted in JSON:

```

<rpc>
<load-configuration format="json">
<configuration-json>
{
  "configuration" : {
    "protocols" : {

```

```

        "ospf" : {
            "@" : {
                "operation" : "delete"
            }
        }
    }
}
</configuration-json>
</load-configuration>
</rpc>

```

Deleting a Configuration Object That Has an Identifier

To delete a configuration object that has an identifier, a client application includes the basic tag elements or configuration statements for its parent levels, as described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#).

When using Junos XML tag elements, the application includes the **delete="delete"** attribute in the opening tag for the object. In the container tag element for the object, it encloses only the identifier tag element, not tag elements that represent any other characteristics of the object. In the following, the identifier tag element is called **<name>**:

```

<configuration>
  <!-- opening tag for each parent of the object -->
    <object delete="delete">
      <name>identifier</name>
    </object>
  <!-- closing tag for each parent of the object -->
</configuration>

```



NOTE: The **delete** attribute appears in the opening container tag, not in the identifier tag element. The presence of the identifier tag element results in the removal of the specified object, not in the removal of the entire hierarchy level represented by the container tag element.

When using formatted ASCII text, the application places the **delete:** statement above the object and its identifier:

```

<configuration-text>
  /* statements for parent levels of the object */
  delete:
    object identifier;
  /* closing braces for parent levels of the object */
</configuration-text>

```

When using configuration mode commands, the application specifies the **delete** command, the statement path to the object, and the object and its identifier.

```

<configuration-set>
  delete statement-path-to-object object identifier
</configuration-set>

```

When using JSON configuration data, the application includes the **"operation" : "delete"** attribute in the attribute list for the object. In the container object, it encloses only the name/value pair representing the identifier. In the following example, the JSON member that specifies the element's identifier has the field name "name":

```
<configuration-json>
{
  "configuration" : {
    /* JSON objects for parent levels of the object */
    "object" : [
      {
        "@" : {
          "operation" : "delete"
        },
        "name" : "identifier"
      }
    ]
    /* closing braces for parent levels of the object */
  }
}
</configuration-json>
```

The following example uses Junos XML tag elements to remove the user object **barbara** from the **[edit system login user]** hierarchy level in the candidate configuration.

Client Application

```
<rpc>
<load-configuration>
<configuration>
<system>
<login>
<user delete="delete">
<name>barbara</name>
</user>
</login>
</system>
</configuration>
</load-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
<load-configuration-results>
<load-success/>
</load-configuration-results>
</rpc-reply>
```

T1138

The following example uses JSON-formatted configuration data to remove the user object **barbara** from the **[edit system login user]** hierarchy level in the candidate configuration.

```
<rpc>
<load-configuration format="json">
<configuration-json>
{
  "configuration" : {
    "system" : {
      "login" : {
        "user" : [
          {
```

```

        "@": {
            "operation": "delete"
        },
        "name": "barbara"
    }
}
}
}
}
}
</configuration-json>
</load-configuration>
</rpc>

```

Deleting a Single-Value or Fixed-Form Option from a Configuration Object

To delete from a configuration object either a fixed-form option or an option that takes just one value, a client application includes the basic tag elements or configuration statements for its parent levels, as described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#). (For information about deleting an option that can take multiple values, see [“Deleting Values from a Multivalue Option of a Configuration Object” on page 192](#).)

When using Junos XML tag elements, the application includes the **delete="delete"** attribute in the empty tag for each option. It does not include tag elements for children that are to remain in the configuration. In the following, the identifier tag element for the object is called **<name>**:

```

<configuration>
  <!-- opening tag for each parent of the object -->
  <object>
    <name>identifier</name> <!-- if object has an identifier -->
    <option1 delete="delete"/>
    <option2 delete="delete"/>
    <!-- tag elements for other options to delete -->
  </object>
  <!-- closing tag for each parent of the object -->
</configuration>

```

When using formatted ASCII text, the application places the **delete:** statement above each option:

```

<configuration-text>
/* statements for parent levels of the object */
  object identifier;
  delete:
    option1;
  delete:
    option2;
/* closing braces for parent levels of the object */
</configuration-text>

```

When using configuration mode commands, the application specifies the **delete** command, the statement path to the option, and the option to be removed. You can specify the full path to the option statement or navigate to the hierarchy level of the object and delete the option statement from that location. Use a separate command to delete each option.

```

<configuration-set>

```

```

    delete statement-path-to-object object identifier option1
    delete statement-path-to-object object identifier option2
</configuration-set>

<configuration-set>
    edit statement-path-to-object object identifier
    delete option1
    delete option2
</configuration-set>

```

When using JSON configuration data to delete an option, the application includes the **"operation": "delete"** attribute in the attribute list for that option. To delete options for a hierarchy level or container object, specify the options to delete at that level.

```

<configuration-json>
{
    "configuration" : {
        /* JSON objects for parent levels */
        "level-or-object" : {
            "@option1" : {
                "operation" : "delete"
            },
            "@option2" : {
                "operation" : "delete"
            }
        }
        /* closing braces for parent levels */
    }
}
</configuration-json>

```

To delete options for an object that has an identifier, include the identifier first, and then specify the options to delete. In the following example, the JSON member that specifies the element's identifier has the field name **"name"**:

```

<configuration-json>
{
    "configuration" : {
        /* JSON objects for parent levels of the object */
        "object" : [
            {
                "name" : "identifier",
                "@option1" : {
                    "operation" : "delete"
                },
                "@option2" : {
                    "operation" : "delete"
                }
            }
        ]
        /* closing braces for parent levels of the object */
    }
}
</configuration-json>

```

The following example shows how to remove the fixed-form **disable** option at the **[edit forwarding-options sampling]** hierarchy level using Junos XML tag elements.

Client Application

```

<rpc>
  <load-configuration>
    <configuration>
      <forwarding-options>
        <sampling>
          <disable delete="delete"/>
        </sampling>
      </forwarding-options>
    </configuration>
  </load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1140

Deleting Values from a Multivalue Option of a Configuration Object

As described in [“Mapping Configuration Statements to Junos XML Tag Elements” on page 19](#), some Junos OS configuration objects are leaf statements that have multiple values. In the formatted ASCII CLI representation, the values are enclosed in square brackets following the name of the object:

```
object [value1 value2 value3 ...];
```

The Junos XML representation does not use a parent tag for the object, but instead uses a separate instance of the object tag element for each value. In the following, the identifier tag element is called **<name>**:

```

<parent-object>
  <name>identifier</name>
  <object>value1</object>
  <object>value2</object>
  <object>value3</object>
</parent-object>

```

To remove one or more values for such an object, a client application includes the basic tag elements or configuration statements for its parent levels, as described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#). When using Junos XML tag elements, the application includes the **delete="delete"** attribute in the opening tag for each value. It does not include tag elements that represent values to be retained. In the following, the identifier tag element for the parent object is called **<name>**:

```

<configuration>
  <!-- opening tag for each parent of the parent object -->
    <parent-object>
      <name>identifier</name>
      <object delete="delete">value1</object>
      <object delete="delete">value2</object>
    </parent-object>
  <!-- closing tag for each parent of the parent object -->
</configuration>

```


When using formatted ASCII text, the application repeats the parent statement for each value and places the **delete:** statement above each paired statement and value:

```
<configuration-text>
  /* statements for parent levels of the parent object */
  parent-object identifier;
  delete:
    object value1;
  delete:
    object value2;
  /* closing braces for parent levels of the parent object */
</configuration-text>
```

When using configuration mode commands, the application specifies the **delete** command, the statement path to each value, and the value to be removed. You can specify the full path to the value or navigate to the hierarchy level of the object and delete the value from that location. Use a separate command to delete each value.

```
<configuration-set>
  delete statement-path-to-parent-object parent-object identifier object
value1
  delete statement-path-to-parent-object parent-object identifier object
value2
</configuration-set>

<configuration-set>
  edit statement-path-to-parent-object parent-object identifier object
  delete value1
  delete value2
</configuration-set>
```

The JSON representation for an object with a multivalue option is a name/value pair where the field name is the object name, and its value, which represents the options, is an array of strings. Junos OS does not support using JSON to delete single values from an object with a multivalue option. To update the option list, you must delete the existing object and then configure a new object with the desired set of values.

The following example shows how to remove two of the permissions granted to the **user-accounts** login class using Junos XML tag elements.

Client Application

```

<rpc>
  <load-configuration>
    <configuration>
      <system>
        <login>
          <class>
            <name>user-accounts</name>
            <permissions delete="delete">configure</permissions>
            <permissions delete="delete">control</permissions>
          </class>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1139

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Renaming Objects In Configuration Data Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to change the name of one or more of a configuration object's identifiers, a client application includes the tag elements described in ["Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol" on page 173](#). When using Junos XML tag elements, the client application includes the **rename="rename"** attribute and an attribute named after the identifier keyword in the object's opening tag. The value of the attribute is the new identifier value. The application includes the identifier tag element to specify the current name. In the following, the identifier tag element is called **<name>**:

```

<configuration>
  <!-- opening tag for each parent of the object -->
    <object rename="rename" name="new-name">
      <name>current-name</name>
    </object>
  <!-- closing tag for each parent of the object -->
</configuration>

```

If the object has multiple identifiers, for each one the application includes both an attribute in the opening tag and an identifier tag element. If one or more of the identifiers is not changing, the attribute value for it is set to its current name. The opening tag appears on two lines for legibility only:

```
<configuration>
  <!-- opening tag for each parent of the object -->
    <object rename="rename" changing-identifier="new-name" \
      unchanging-identifier="current-name">
      <changing-identifier>current-name</changing-identifier>
      <unchanging-identifier>current-name</unchanging-identifier>
    </object>
  <!-- closing tag for each parent of the object -->
</configuration>
```

When using configuration mode commands to rename an object, the application specifies the **rename** command equivalent to the CLI configuration mode command. If the object has multiple identifiers, the application includes a separate **rename** command for each identifier.

```
<configuration-set>
  rename statement-path-to-object object current-name to object new-name
</configuration-set>
```



NOTE: The rename operation is not available when formatted ASCII text or JSON is used to represent the configuration data.

For Junos XML tag elements the **rename** attribute can be combined with the **inactive** or **active** attribute to deactivate or reactivate the configuration element as it is renamed. For more information, see [“Changing a Configuration Element's Activation State Simultaneously with Other Changes Using the Junos XML Protocol”](#) on page 208.

The following example shows how to change the name of a firewall filter from **access-control** to **new-access-control** using Junos XML tag elements. This operation is equivalent to the following configuration mode command:

```
[edit firewall family inet]
user@host# rename filter access-control to filter new-access-control
```

Client Application

```

<rpc>
  <load-configuration>
    <configuration>
      <firewall>
        <family>
          <inet>
            <filter rename="rename" name="new-access-control">
              <name>access-control</name>
            </filter>
          </inet>
        </family>
      </firewall>
    </configuration>
  </load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1143

The following example shows how to change the name of a firewall filter from **access-control** to **new-access-control** using configuration mode commands:

```

<rpc>
  <load-configuration action="set" format="text">
    <configuration-set>
      rename firewall family inet filter access-control to filter
new-access-control
    </configuration-set>
  </load-configuration>
</rpc>

```

The following example shows how to change the identifiers for an OSPF virtual link (defined at the **[edit protocols ospf area area]** hierarchy level) from **neighbor-id 192.168.0.3** and **transit-area 10.10.10.1** to **neighbor-id 192.168.0.7** and **transit-area 10.10.10.5**. This operation is equivalent to the following configuration mode command (which appears on two lines for legibility only):

```

[edit protocols ospf area area]
user@host# rename virtual-link neighbor-id 192.168.0.3 transit-area \
10.10.10.1 to virtual-link neighbor-id 192.168.0.7 transit-area 10.10.10.5

```

Client Application

```

<rpc>
  <load-configuration>
    <configuration>
      <protocols>
        <ospf>
          <area>
            <name>area</name>
            <virtual-link rename="rename" neighbor-id="192.168.0.7
transit-area="10.10.10.5">
              <neighbor-id>192.168.0.3</neighbor-id>
              <transit-area>10.10.10.1</transit-area>
            </virtual-link>
          </area>
        </ospf>
      </protocols>

```

```

    </configuration>
  </load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Reordering Elements In Configuration Data Using the Junos XML Protocol

For most configuration objects, the order in which the object or its children are created is not significant, because the Junos OS configuration management software stores and displays configuration objects in predetermined positions in the configuration hierarchy. However, some configuration objects—such as routing policies and firewall filters—consist of elements that must be processed and analyzed sequentially in order to produce the intended routing behavior. When a client application uses the Junos XML management protocol to add a new element to an ordered set, the element is appended to the existing list of elements. The client application can then reorder the elements, if appropriate.

In a Junos XML protocol session with a device running Junos OS, to change the order of configuration elements in an ordered set, a client application first includes the tag elements described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#). If using Junos XML tag elements, the application emits the container tag element that represents the ordered set, and encloses the tag element for each identifier of the configuration element that is moving. In the following examples, the identifier tag element is called **<name>**.

To move an existing element to the first position in an ordered set, the application includes the **insert="first"** attribute in the opening container tag for that element.

```

<configuration>
  <!-- opening tag for each parent of the set -->
    <ordered-set insert="first">
      <name>identifier-for-moving-object</name>
    </ordered-set>
  <!-- closing tag for each parent of the set -->
</configuration>

```

To move an existing element to a position that is relative to another element, the application includes the **insert="before"** or **insert="after"** attribute in the opening container tag to indicate the new position of the moving element relative to another reference element in the set. To identify the reference element, it includes each of the reference element's identifiers as an attribute in the opening container tag for the ordered set.

In the following example, the elements in the set have one identifier, called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the set -->
    <ordered-set insert="(before | after)" name="referent-value">
      <name>identifier-for-moving-object</name>
    </ordered-set>
  <!-- closing tag for each parent of the set -->
</configuration>
```

In the following example, each element in the set has two identifiers. The opening tag appears on two lines for legibility only:

```
<configuration>
  <!-- opening tag for each parent of the set -->
    <ordered-set insert="(before | after)" identifier1="referent-value"
      identifier2="referent-value">
      <identifier1>value-for-moving-object</identifier1>
      <identifier2>value-for-moving-object</identifier2>
    </ordered-set>
  <!-- closing tag for each parent of the set -->
</configuration>
```

The **insert** attribute can be combined with the **inactive** or **active** attribute to deactivate or reactivate the configuration element as it is reordered. For more information, see [“Changing a Configuration Element’s Activation State Simultaneously with Other Changes Using the Junos XML Protocol” on page 208](#).

When using configuration mode commands to reorder elements, the application specifies the **insert** command that is equivalent to the CLI configuration mode command.

```
<configuration-set>
  insert statement-path-to-object identifier-for-moving-object (before |
after) referent-value
</configuration-set>
```



NOTE: The **insert="first"** attribute has no equivalent CLI configuration mode command.



NOTE: The reordering operation is not available when formatted ASCII text or JSON is used to represent the configuration data.

The following example shows how to move a firewall filter called **older-filter**, defined at the **[edit firewall filter]** hierarchy level, and place it after another filter called **newer-filter** using Junos XML tag elements. This operation is equivalent to the following configuration mode command:

```
[edit]
```

```
user@host# insert firewall family inet filter older-filter after filter newer-filter
```

Client Application**Junos XML Protocol Server**

```
<rpc>
  <load-configuration>
    <configuration>
      <firewall>
        <family>
          <inet>
            <filter insert="after" name="newer-filter">
              <name>older-filter</name>
            </filter>
          </inet>
        </family>
      </firewall>
    </configuration>
  </load-configuration>
</rpc>

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1141

The following example shows how to move a firewall filter called **older-filter**, defined at the **[edit firewall filter]** hierarchy level, and place it after another filter called **newer-filter** using configuration mode commands:

```
<rpc>
  <load-configuration action="set" format="text">
    <configuration-set>
      insert firewall family inet filter older-filter after filter
      newer-filter
    </configuration-set>
  </load-configuration>
</rpc>
```

The following example shows how to move an OSPF virtual link defined at the **[edit protocols ospf area area]** hierarchy level. The link with identifiers **neighbor-id 192.168.0.3** and **transit-area 10.10.10.1** moves before the link with identifiers **neighbor-id 192.168.0.5** and **transit-area 10.10.10.2**. This operation is equivalent to the following configuration mode command:

```
[edit protocols ospf area area]
user@host# insert virtual-link neighbor-id 192.168.0.3 transit-area 10.10.10.1
before virtual-link neighbor-id 192.168.0.5 transit-area 10.10.10.2
```

Client Application

```
<rpc>
  <load-configuration>
    <configuration>
      <protocols>
        <ospf>
          <area>
            <name>area</name>
            <virtual-link insert="before" neighbor-id="192.168.0.5"
transit-area="10.10.10.2">
              <neighbor-id>192.168.0.3</neighbor-id>
              <transit-area>10.10.10.1</transit-area>
            </virtual-link>
          </area>
        </ospf>
      </protocols>
    </configuration>
  </load-configuration>
</rpc>
```

```
        </area>
      </ospf>
    </protocols>
  </configuration>
</load-configuration>
</rpc>
```

**Junos XML Protocol
Server**

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

**Related
Documentation**

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Protecting or Unprotecting a Configuration Object Using the Junos XML Protocol

The **protect** attribute prevents changes to selected configuration hierarchies and statements. You cannot alter a protected element either manually through the CLI or automatically using commit scripts or remote procedure calls. If you attempt to make configuration changes to a protected statement or within a protected hierarchy, Junos OS issues a warning, and the configuration change fails.

If a configuration hierarchy or statement is protected, users cannot perform the following activities:

- Delete or modify the hierarchy or a statement or identifier within the protected hierarchy (Deletion of an unprotected hierarchy that contains protected elements deletes all unprotected child elements and preserves all protected child elements.)
- Insert a new configuration statement or an identifier within the protected hierarchy
- Rename the protected statement or a statement or identifier within the protected hierarchy
- Copy a configuration into the protected hierarchy
- Activate or deactivate the protected statements or statements within the protected hierarchy
- Annotate the protected statement or hierarchy, or statements within the protected hierarchy

If you protect a configuration statement or hierarchy that does not exist, Junos OS first creates the configuration element and then protects it. If you unprotect a statement or element that is not protected, no action is taken.

You can identify protected elements when you display the configuration. If you display the configuration in text format, protected elements are preceded by **protect:**. If you display the configuration in XML format, the opening tag of the protected element contains the **protect="protect"** attribute. If you display the configuration as configuration mode commands, the **protect** commands indicate protected elements. If you display the configuration in JSON, protected hierarchies and statements include the **"protect" : true** attribute in their attribute list.



NOTE: A user or client application must have permission to modify the configuration in order to protect or unprotect configuration objects.

In a Junos XML protocol session with a device running Junos OS, to protect a configuration element from changes or to unprotect a previously protected element, a client application first includes the tag elements described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#). When using Junos XML tag elements to represent the configuration, the client application includes the **protect="protect"** or **unprotect="unprotect"** attribute in the object's opening tag. The application includes any necessary identifier tag element. In the following sample RPC, the identifier tag element is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the object -->
    <object (protect="protect" | unprotect="unprotect")>
      <name>identifier</name>
    </object>
  <!-- closing tag for each parent of the object -->
</configuration>
```

When using formatted ASCII text to protect or unprotect an object, the application precedes the element with the **protect:** or **unprotect:** operator as appropriate. If you are protecting a hierarchy level and no additional child elements under that hierarchy, add a semicolon after the element statement.

```
<configuration-text>
  /* statements for parent levels */

  /* For an object with an identifier */
  (protect: | unprotect:)
  object identifier {
    /* Child configuration statements */
  }

  /* For a hierarchy level or object without an identifier */
  (protect: | unprotect:)
  element {
    /* Child configuration statements */
  }

  /* closing braces for parent levels */
</configuration-text>
```

When using configuration mode commands to protect an object, the application specifies the **protect** or **unprotect** command equivalent to the CLI configuration mode command. You can protect both hierarchies and individual statements.

```
<configuration-set>
  (protect | unprotect) statement-path-to-hierarchy
  (protect | unprotect) statement-path-to-object object identifier
</configuration-set>
```

When using JSON configuration data to represent the configuration, the client application protects or unprotects an object by including the appropriate attribute in the attribute list of the object. The client includes the **"protect" : true** attribute to protect the object and includes either the **"protect" : false** or **"unprotect" : true** attribute to unprotect the object. To protect or unprotect an object that has an identifier, the client also includes the identifier for the object.

The following generic JSON configuration indicates the placement of the attribute when protecting a hierarchy, an object that has an identifier, and a leaf statement.

```
<configuration-json>
{
  "configuration" : {
    /* JSON objects for parent hierarchies */
    "hierarchy" : {
      "@" : {
        "comment" : "/* protect a hierarchy */" ,
        "protect" : true
      },
      "object" : [
        {
          "@" : {
            "comment" : "/* protect an object with an identifier
*/" ,
            "protect" : true
          },
          "name" : "identifier",
          "@statement-name" : {
            "comment" : "/* protect a statement */" ,
            "protect" : true
          }
        }
      ]
    }
    /* closing braces for parent hierarchies */
  }
}
</configuration-json>
```



NOTE: In Junos OS configuration data that is represented using JSON, the value for the **"protect"** and **"unprotect"** attribute is type Boolean, which is expressed in lowercase and is not enclosed in quotes.

The following example protects the **[edit access]** hierarchy level of the configuration using Junos XML tag elements:

```
<rpc>
  <load-configuration>
```

```

    <configuration>
      <access protect="protect"/>
    </configuration>
  </load-configuration>
</rpc>

```

Once protected, any attempt to modify the **[edit access]** hierarchy level produces a warning. The following RPC attempts to delete the **[edit access]** hierarchy level. Because that hierarchy level is protected, the server returns a warning that the hierarchy is protected, and the configuration change fails.

```

<rpc>
  <load-configuration>
    <configuration>
      <access delete="delete"/>
    </configuration>
  </load-configuration>
</rpc>

<xnm:warning xmlns="http://xml.juniper.net/xnm/1.1/xnm"
xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
  <message>
    [access] is protected, 'access' cannot be deleted
  </message>
</xnm:warning>

```

Related Documentation

- [Example: Protecting the Junos OS Configuration from Modification or Deletion](#)
- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)

Changing a Configuration Element's Activation State Using the Junos XML Protocol

When a configuration element (hierarchy level or configuration object) is deactivated and the configuration is committed, the deactivated element remains in the configuration, but the element does not affect the functioning of the device. Deactivating configuration elements is useful when you want to troubleshoot issues by suppressing the behavior of a configuration element without deleting it from the configuration. Additionally, you can configure and deactivate new configuration elements to prepare the configuration to accommodate new hardware before it is available.

In a Junos XML protocol session with a device running Junos OS, a client application can deactivate an existing element or simultaneously create and deactivate a new element. A client application can also activate a deactivated element so that when the configuration is committed, the element again has an effect on the functioning of the

device. The following sections discuss how to create and deactivate new configuration elements and how to activate or deactivate existing elements:

- [Deactivating a Newly Created Element on page 204](#)
- [Deactivating or Reactivating an Existing Element on page 205](#)

Deactivating a Newly Created Element

To create an element and immediately deactivate it, a client application first includes the basic tag elements or configuration statements for the new element and any child elements as described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#).

When using Junos XML tag elements to create and deactivate a new element, the application includes the **inactive="inactive"** attribute in the opening tag for the new element. In the following example, the identifier tag element is called **<name>**:

```
<configuration>
  <!-- opening tag for each parent of the element -->
  <element inactive="inactive">
    <name>identifier</name> <!-- if element has an identifier -->
    <!-- tag elements for each child of the element -->
  </element>
  <!-- closing tag for each parent of the element -->
</configuration>
```

When using formatted ASCII text to create and deactivate a new element, the application precedes the new element with the **inactive:** operator.

```
<configuration-text>
/* statements for parent levels */

/* For an object with an identifier */
inactive:
object identifier {
  /* Child configuration statements */
}

/* For a hierarchy level or object without an identifier */
inactive:
element {
  /* Child configuration statements */
}

/* closing braces for parent levels */
</configuration-text>
```

When using configuration mode commands to create an inactive element, the application first creates the element with the **set** command and then deactivates it by using the **deactivate** command.

```
<configuration-set>
  set statement-path-to-object object identifier
  deactivate statement-path-to-object object identifier
</configuration-set>
```

When using JSON configuration data to create and deactivate a new element, the client application includes the `"inactive" : true` attribute in the attribute list for that element. The following generic JSON configuration indicates the placement of the attribute for deactivating a hierarchy or container object, an object that has an identifier, and a leaf statement.

```
<configuration-json>
{
  "configuration" : {
    /* JSON objects for parent levels */
    "level-or-container" : {
      "@" : {
        "comment" : "/* deactivate a hierarchy */",
        "inactive" : true
      },
      "object" : [
        {
          "@" : {
            "comment" : "/* deactivate an object with an identifier
*/",
            "inactive" : true
          },
          "name" : "identifier",
          "statement-name" : "statement-value",
          "@statement-name" : {
            "comment" : "/* deactivate a statement */",
            "inactive" : true
          },
          /* additional data and child objects */ # if any
        }
      ]
    }
  }
  /* closing braces for parent levels */
}
</configuration-json>
```

Deactivating or Reactivating an Existing Element

To deactivate an existing element, or activate a previously deactivated element, a client application includes the basic tag elements or configuration statements for its parent levels, as described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#).

When using Junos XML tag elements to represent a configuration object that has an identifier, the application includes the `inactive="inactive"` or `active="active"` attribute in the object's opening container tag and also emits the identifier tag element and value. In the following example, the identifier tag element is called `<name>`. To represent a hierarchy level or container object that has children but does not have an identifier, the application uses an empty tag:

```
<configuration>
  <!-- opening tag for each parent of the element -->
  <!-- - For an object with an identifier -->
  <object ( inactive="inactive" | active="active" ) >
    <name>identifier</name>
  </object>

  <!-- For a hierarchy level or object without an identifier -->
```

```

        <level-or-container ( inactive="inactive" | active="active" ) />
        <!-- closing tag for each parent of the element -->
    </configuration>

```

When using formatted ASCII text to represent the element, the application precedes the element with the **inactive:** or **active:** operator. The name of a hierarchy level or container object is followed by a semicolon (even though in the existing configuration it is followed by curly braces that enclose its child statements):

```

<configuration-text>
/* statements for parent levels */

/* For an object with an identifier */
(inactive | active):
object identifier;

/* For a hierarchy level or object without an identifier */
(inactive | active):
object-or-level;

/* closing braces for parent levels */
</configuration-text>

```

When using configuration mode commands to activate or deactivate an object, the application specifies the **activate** or **deactivate** command equivalent to the CLI configuration mode command.

```

<configuration-set>
/* For an object with an identifier */
activate statement-path-to-object object identifier
deactivate statement-path-to-object object identifier

/* For a hierarchy level or object without an identifier */
activate statement-path-to-object-or-level object-or-level
deactivate statement-path-to-object-or-level object-or-level
</configuration-set>

```

When using JSON to represent the element, the client application activates or deactivates the element by including the **"active": true** or **"inactive": true** attribute, respectively, in the attribute list of that element. The following generic JSON configuration indicates the placement of the attribute for activating or deactivating existing hierarchies or container objects, objects that have an identifier, and leaf statements.

```

<configuration-json>
{
    "configuration" : {
        /* JSON objects for parent levels */
        "level-or-container" : {
            "@" : {
                "comment" : "/* activate or deactivate a hierarchy */",
                "(active | inactive)" : true
            },
            "object" : [
                {
                    "@" : {
                        "comment" : "/* activate or deactivate an object with
an identifier */",
                        "(active | inactive)" : true
                    },
                    "name" : "identifier",
                    "@statement-name" : {

```

```

        "comment" : "/* activate or deactivate a statement
*/",
        "(active | inactive)" : true
    }
}
]
}
/* closing braces for parent levels */
}
}
</configuration-json>

```

The following example shows how to deactivate the **isis** hierarchy level at the **[edit protocols]** hierarchy level in the candidate configuration using Junos XML tag elements.

Client Application

```

<rpc>
  <load-configuration>
    <configuration>
      <protocols>
        <isis inactive="inactive"/>
      </protocols>
    </configuration>
  </load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1145

The following example shows how to deactivate the **isis** hierarchy level at the **[edit protocols]** hierarchy level in the candidate configuration using JSON.

```

<rpc>
<load-configuration format="json">
<configuration-json>
{
  "configuration" : {
    "protocols" : {
      "isis" : {
        "@": {
          "inactive" : true
        }
      }
    }
  }
}
</configuration-json>
</load-configuration>
</rpc>

```

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)

- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)
- [Changing a Configuration Element's Activation State Simultaneously with Other Changes Using the Junos XML Protocol on page 208](#)

Changing a Configuration Element's Activation State Simultaneously with Other Changes Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, a client application can deactivate or reactivate an element at the same time it performs other operations on it (except deletion), by combining the appropriate attributes or operators with the **inactive** or **active** attribute or operator. For basic information about activating or deactivating an element, see [“Changing a Configuration Element's Activation State Using the Junos XML Protocol” on page 203](#).

To define the element to deactivate or activate, a client application includes the basic tag elements or configuration statements for its parent levels, as described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#). When using Junos XML tag elements to represent the element, the application includes the **inactive="inactive"** or **active="active"** attribute along with the other appropriate attributes in the **<load-configuration>** tag. When using formatted ASCII text, the application combines the **inactive** or **active** operator with the other operator.

For instructions, see the following sections:

- [Replacing an Element and Setting Its Activation State on page 208](#)
- [Reordering an Element and Setting Its Activation State on page 210](#)
- [Renaming an Object and Setting Its Activation State on page 210](#)
- [Example: Replacing an Object and Deactivating It on page 211](#)

Replacing an Element and Setting Its Activation State

To replace (completely reconfigure) an element and simultaneously deactivate or activate it, a client application includes the tag elements or statements that represent all of the element's characteristics (for complete information about the syntax for defining elements, see [“Replacing Elements in Configuration Data Using the Junos XML Protocol” on page 181](#)). The client application uses the attributes and operators discussed in the following examples to indicate which element is being replaced and activated or deactivated.

Using Junos XML Tag Elements for the Replacement Element

If using Junos XML tag elements to represent the element, a client application includes the **action="replace"** attribute in the **<load-configuration>** tag element:

```
<rpc>
  <!-- For a file -->
```



```

<load-configuration action="replace" url="file"/>

<!-- For a data stream -->
<load-configuration action="replace">
  <!-- Junos XML tag elements -->
</load-configuration>
</rpc>

```

In the opening tag for the replacement element, the application includes two attributes—the **replace="replace"** attribute and either the **inactive="inactive"** or **active="active"** attribute. It includes tag elements for all children being defined for the element. In the following, the identifier tag element is called **<name>**:

```

<configuration>
  <!-- opening tag for each parent of the element -->
  <element replace="replace" (inactive="inactive" | active="active")>
    <name>identifier</name> <!-- if element has an identifier -->
    <!-- tag elements for each child of the element -->
  </element>
  <!-- closing tag for each parent of the element -->
</configuration>

```

Using Formatted ASCII Text for the Replacement Element

If using formatted ASCII text to represent the element, a client application includes the **action="replace"** and **format="text"** attributes in the **<load-configuration>** tag:

```

<rpc>
  <!-- For a file -->
  <load-configuration action="replace" format="text" url="file"/>

  <!-- For a data stream -->
  <load-configuration action="replace" format="text">
    <!-- formatted ASCII configuration statements -->
  </load-configuration>
</rpc>

```

The application places the **inactive:** or **active:** operator on the line above the new element and the **replace:** operator directly before the new element. It includes all child statements that it is defining for all children of the element:

```

<configuration-text>
/* statements for parent levels */

/* For an object with an identifier */
(inactive | active):
replace: object identifier {
  /* Child configuration statements */
}

/* For a hierarchy level or object without an identifier */
(inactive | active):
replace: element {
  /* Child configuration statements */
}

/* closing braces for parent levels */
</configuration-text>

```

Reordering an Element and Setting Its Activation State

To reorder an element in an ordered list and simultaneously deactivate or activate it, the application combines the **insert** attribute and identifier attribute for the reference element with the **inactive** or **active** attribute. In the following, the identifier tag element for the moving element is called **<name>**. The opening tag appears on two lines for legibility only:

```
<configuration>
  <!-- opening tag for each parent of the set -->
    <ordered-set insert="( before | after )" reference-identifier="value"
      (inactive="inactive" | active="active")>
      <name>identifier-for-moving-object</name>
    </ordered-set>
  <!-- closing tag for each parent of the set -->
</configuration>
```



NOTE: The reordering operation is not available when formatted ASCII text is used to represent the configuration data.

For complete information about reordering elements, see ["Reordering Elements In Configuration Data Using the Junos XML Protocol"](#) on page 197.

Renaming an Object and Setting Its Activation State

To rename an object (change the value of one or more of its identifiers) and simultaneously deactivate or activate it, the application combines the **rename** attribute and identifier attribute for the new name with the **inactive** or **active** attribute.

If the object has one identifier (here called **<name>**), the syntax is as follows (the opening tag appears on two lines for legibility only):

```
<configuration>
  <!-- opening tag for each parent of the object -->
    <object rename="rename" name="new-name" \
      (inactive="inactive" | active="active")>
      <name>current-name</name>
    </object>
  <!-- closing tag for each parent of the object -->
</configuration>
```

If the object has multiple identifiers and only one is changing, the syntax is as follows (the opening tag appears on multiple lines for legibility only):

```
<configuration>
  <!-- opening tag for each parent of the object -->
    <object rename="rename" changing-identifier="new-name" \
      unchanging-identifier="current-name" \
      (inactive="inactive" | active="active")>
      <changing-identifier>current-name</changing-identifier>
      <unchanging-identifier>current-name</unchanging-identifier>
    </object>
  <!-- closing tag for each parent of the object -->
</configuration>
```



NOTE: The renaming operation is not available when formatted ASCII text is used to represent the configuration data.

For complete information about renaming elements, see “Renaming Objects In Configuration Data Using the Junos XML Protocol” on page 194.

Example: Replacing an Object and Deactivating It

The following example shows how to replace the information at the [edit protocols bgp] hierarchy level in the candidate configuration for the group called G3, and also deactivate the group so that it is not used in the actual configuration when the candidate is committed:

Client Application

```
<rpc>
  <load-configuration action="replace">
    <configuration>
      <protocols>
        <bgp>
          <group replace="replace" inactive="inactive">
            <name>G3</name>
            <type>external</type>
            <peer-as>58</peer-as>
            <neighbor>
              <name>10.0.20.1</name>
            </neighbor>
          </group>
        </bgp>
      </protocols>
    </configuration>
  </load-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1146

The following example shows how to use formatted ASCII text to make the same changes:

Client Application

```

<rpc>
  <load-configuration action="replace" format="text">
    <configuration-text>
      protocols {
        bgp {
          replace:
            inactive: group G3 {
              type external;
              peer-as 58;
              neighbor 10.0.20.1;
            }
        }
      }
    </configuration-text>
  </load-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>

```

T1147

Related Documentation

- [Requesting Configuration Changes Using the Junos XML Protocol on page 160](#)
- [Uploading and Formatting Configuration Data in a Junos XML Protocol Session on page 162](#)
- [Specifying the Scope of Configuration Data to Upload in a Junos XML Protocol Session on page 169](#)
- [Replacing the Candidate Configuration Using the Junos XML Protocol on page 169](#)
- [Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol on page 173](#)
- [Changing a Configuration Element's Activation State Using the Junos XML Protocol on page 203](#)

Replacing Patterns in Configuration Data Using the NETCONF or Junos XML Protocol

Starting in Junos OS Release 15.1R1, in a NETCONF or Junos XML protocol session with a device running Junos OS, you can replace variables and identifiers in the configuration by including the **replace-pattern** attribute when performing a **<load-configuration>** operation. The **replace-pattern** attribute replaces the given pattern with another pattern either globally or at the indicated hierarchy or object level in the configuration. For example, you can use this feature to find and replace all occurrences of an interface name when a PIC is moved to another slot in the router. The functionality of the attribute is identical to that of the **replace pattern** configuration mode command in the Junos OS CLI.



NOTE: The replace pattern operation can only be used with configuration data formatted as Junos XML tag elements.

To replace a pattern, a client application emits the `<rpc>` and `<load-configuration>` tag elements and includes the basic Junos XML tag elements described in [“Creating, Modifying, or Deleting Configuration Elements Using the Junos XML Protocol” on page 173](#). At the hierarchy or object level where the pattern should be replaced, the client includes the **replace-pattern** attribute, which specifies the pattern to replace, the **with** attribute, which specifies the replacement pattern, and optionally includes the **upto** attribute, which indicates the number of occurrences to replace. If the **upto** attribute is omitted or set to zero, Junos OS replaces all instances of the pattern within the specified scope. The placement of the attributes within the configuration determines the scope as described in the following sections:

- [Replacing Patterns Globally Within the Configuration on page 213](#)
- [Replacing Patterns Within a Hierarchy Level or Container Object That Has No Identifier on page 214](#)
- [Replacing Patterns for a Configuration Object That Has an Identifier on page 215](#)

Replacing Patterns Globally Within the Configuration

To globally replace a pattern throughout the candidate configuration or open configuration database, include the **replace-pattern** and **with** attributes in the opening `<configuration>` tag.

```
<rpc>
  <load-configuration>
    <configuration replace-pattern="pattern1" with="pattern2" [upto="n"]>
    </configuration>
  </load-configuration>
</rpc>
```

For example, the following RPC replaces all instances of 172.17.1.5 with 172.16.1.1:

```
<rpc>
  <load-configuration>
    <configuration replace-pattern="172.17.1.5" with="172.16.1.1">
    </configuration>
  </load-configuration>
</rpc>
```

After executing the RPC, you can compare the updated candidate configuration to the active configuration to verify the pattern replacement. You must commit the configuration for the changes to take effect.

```
<rpc>
  <get-configuration compare="rollback" rollback="0" format="text">
  </get-configuration>
</rpc>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
  <configuration-information>
    <configuration-output>
      [edit groups global system ntp]
      -   boot-server 172.17.1.5;
      +   boot-server 172.16.1.1;
      [edit groups global system ntp]
```

```
+    server 172.16.1.1;
-    server 172.17.1.5;
</configuration-output>
</configuration-information>
</rpc-reply>
```

Replacing Patterns Within a Hierarchy Level or Container Object That Has No Identifier

To replace a pattern under a specific hierarchy level including all of its children (or a container object that has children but no identifier), a client application includes the **replace-pattern** and **with** attributes in the empty tag that represents the hierarchy level or container object.

```
<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent element -->
      <level-or-object replace-pattern="pattern1" with="pattern2" [upto="n"]/>
      <!-- closing tag for each parent element -->
    </configuration>
  </load-configuration>
</rpc>
```

The following RPC replaces instances of fe-0/0/1 with ge-1/0/1 at the **[edit interfaces]** hierarchy level:

```
<rpc>
  <load-configuration>
    <configuration>
      <interfaces replace-pattern="fe-0/0/1" with="ge-1/0/1"/>
    </configuration>
  </load-configuration>
</rpc>
```

After executing the RPC, you can compare the updated candidate configuration to the active configuration to verify the pattern replacement. For example:

```
<rpc>
  <get-configuration compare="rollback" rollback="0" format="text">
</get-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
  <configuration-information>
    <configuration-output>
      [edit interfaces]
      - fe-0/0/1 {
      -   unit 0 {
      -     family inet {
      -       address 10.0.1.1/27;
      -     }
      -   }
      + ge-1/0/1 {
      +   unit 0 {
      +     family inet {
      +       address 10.0.1.1/27;
```

```

+      }
+    }
+  }
</configuration-output>
</configuration-information>
</rpc-reply>

```

Replacing Patterns for a Configuration Object That Has an Identifier

To replace a pattern for a configuration object that has an identifier, a client application includes the **replace-pattern** and **with** attributes in the opening tag for the object, which then encloses the identifier tag element for that object. In the following example, the identifier tag element is **<name>**:

```

<rpc>
  <load-configuration>
    <configuration>
      <!-- opening tag for each parent element -->
      <container-tag replace-pattern="pattern1" with="pattern2" [upto="n"]>
        <name>identifier</name>
      </container-tag>
      <!-- closing tag for each parent element -->
    </configuration>
  </load-configuration>
</rpc>

```

The following RPC replaces instances of "4.5" with "4.1", but only for the fe-0/0/2 interface under the **[edit interfaces]** hierarchy:

```

<rpc>
  <load-configuration>
    <configuration>
      <interfaces>
        <interface replace-pattern="4.5" with="4.1">
          <name>fe-0/0/2</name>
        </interface>
      </interfaces>
    </configuration>
  </load-configuration>
</rpc>

```

After executing the RPC, you can compare the updated candidate configuration to the active configuration to verify the pattern replacement. For example:

```

<rpc>
  <get-configuration compare="rollback" rollback="0" format="text">
  </get-configuration>
</rpc>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
  <configuration-information>
    <configuration-output>
      [edit interfaces fe-0/0/2 unit 0 family inet]
      +      address 10.0.4.1/30;
      -      address 10.0.4.5/30;
    </configuration-output>
  </configuration-information>

```

- Related Documentation**
- [replace-pattern on page 153](#)
 - *Using Global Replace in the Junos OS Configuration*
 - *Common Regular Expressions to Use with the replace Command*
 - *replace*

CHAPTER 12

Committing the Configuration on a Device Using the Junos XML Protocol

- [Verifying Configuration Syntax Using the Junos XML Protocol on page 217](#)
- [Committing the Candidate Configuration Using the Junos XML Protocol on page 218](#)
- [Committing a Private Copy of the Configuration Using the Junos XML Protocol on page 219](#)
- [Committing a Configuration at a Specified Time Using the Junos XML Protocol on page 221](#)
- [Committing the Candidate Configuration Only After Confirmation Using the Junos XML Protocol on page 223](#)
- [Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol on page 225](#)
- [Logging a Message About a Commit Operation Using the Junos XML Protocol on page 233](#)
- [Viewing the Configuration Revision Identifier for Determining Synchronization Status of Devices with NMS on page 235](#)

Verifying Configuration Syntax Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, during the process of committing the candidate configuration or a private copy, the Junos XML protocol server first confirms that the candidate configuration is syntactically correct. If the syntax check fails, the server does not commit the configuration. To avoid the potential complications of such a failure, it often makes sense to confirm the correctness of the candidate configuration before actually committing it.

To verify the syntax of the candidate configuration prior to committing it, a client application encloses an empty `<check/>` tag in the `<commit-configuration>` and `<rpc>` tag elements.

```
<rpc>
  <commit-configuration>
    <check/>
  </commit-configuration>
</rpc>
```

The Junos XML protocol server encloses its response in `<rpc-reply>`, `<commit-results>`, and `<routing-engine>` tag elements. If the candidate configuration syntax is valid, the `<routing-engine>` tag element encloses the `<commit-check-success/>` tag and the `<name>` tag element, which reports the name of the Routing Engine on which the check succeeded (re0 on routing platforms that use a single Routing Engine, and either re0 or re1 on routing platforms that can have two Routing Engines).

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>(re0 | re1)</name>
      <commit-check-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

If the candidate configuration syntax is not valid, the server returns an `<xnm:error>` element, which encloses child tag elements that describe the error.

The `<check/>` tag can be combined with the `<synchronize/>` tag to verify the check the syntactic correctness of a local configuration on both Routing Engines.

**Related
Documentation**

- [Committing the Candidate Configuration Using the Junos XML Protocol on page 218](#)
- [Committing a Private Copy of the Configuration Using the Junos XML Protocol on page 219](#)
- [<commit-configuration> on page 104](#)

Committing the Candidate Configuration Using the Junos XML Protocol

When you commit the candidate configuration on a device running Junos OS, it becomes the active configuration on the routing, switching, or security platform. In a Junos XML protocol session, to commit the candidate configuration, a client application encloses the empty `<commit-configuration/>` tag in an `<rpc>` tag element.

```
<rpc>
  <commit-configuration/>
</rpc>
```

We recommend that the client application lock the candidate configuration before modifying it and emit the `<commit-configuration/>` tag while the configuration is still locked. This process avoids inadvertently committing changes made by other users or applications. After committing the configuration, the application must unlock it in order for other users and applications to make changes. For instructions, see [“Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol” on page 86](#).

The Junos XML protocol server reports the results of the commit operation in `<rpc-reply>`, `<commit-results>`, and `<routing-engine>` tag elements. If the commit operation succeeds, the `<routing-engine>` tag element encloses the `<commit-success/>` tag and the `<name>` tag element, which reports the name of the Routing Engine on which the commit operation succeeded (re0 on devices that use a single Routing Engine, and either re0 or re1 on devices that can have two Routing Engines).

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>(re0 | re1)</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

If the commit operation fails, the server returns an `<xnm:error>` tag element, which encloses child tag elements that describe the error. The most common causes of failure are semantic or syntactic errors in the candidate configuration.

Related Documentation

- [Verifying Configuration Syntax Using the Junos XML Protocol on page 217](#)
- [Committing a Private Copy of the Configuration Using the Junos XML Protocol on page 219](#)
- [Committing a Configuration at a Specified Time Using the Junos XML Protocol on page 221](#)
- [Committing the Candidate Configuration Only After Confirmation Using the Junos XML Protocol on page 223](#)
- [Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol on page 225](#)
- [Logging a Message About a Commit Operation Using the Junos XML Protocol on page 233](#)
- [<commit-configuration> on page 104](#)

Committing a Private Copy of the Configuration Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to commit a private copy of the configuration so that it becomes the active configuration on the routing, switching, or security platform, a client application encloses the empty `<commit-configuration/>` tag in an `<rpc>` tag element (just as for the candidate configuration).

```
<rpc>
  <commit-configuration/>
</rpc>
```

The Junos XML protocol server creates a copy of the current candidate configuration, merges in the changes made to the private copy of the configuration, and then commits the combined candidate to make it the active configuration on the device. The server reports the results of the commit operation in `<rpc-reply>` and `<commit-results>` tag elements.

If the private copy does not include any changes, the server emits the opening `<commit-results>` tag and closing `</commit-results>` tags with nothing between.

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
  </commit-results>
</rpc-reply>
```

If the private copy includes changes and the commit operation succeeds, the server emits the `<load-success/>` tag when it merges the changes in the private copy into the candidate configuration. The `<routing-engine>` element encloses the `<commit-success/>` tag and the `<name>` element, which reports the name of the Routing Engine on which the commit operation succeeded (re0 on devices that use a single Routing Engine, and either re0 or re1 on devices that can have two Routing Engines).

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <load-success/>
    <routing-engine>
      <name>(re0 | re1)</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

If the private copy includes changes that conflict with the regular candidate configuration, the commit fails. The `<load-error-count>` tag element reports the number of errors and an `<xnm:error>` tag element encloses tag elements that describe the error.

There are restrictions on committing a private copy. For example, the commit fails if the regular candidate configuration is locked by another user or application, or if it includes uncommitted changes made since the private copy was created. For more information, see the *CLI User Guide*.

Most of the variants of the commit operation are available for a private copy.

- Scheduling the commit for a later time, as described in [“Committing a Configuration at a Specified Time Using the Junos XML Protocol” on page 221](#).
- Synchronizing the configuration on both Routing Engines, as described in [“Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol” on page 225](#).
- Logging a commit-time message, as described in [“Logging a Message About a Commit Operation Using the Junos XML Protocol” on page 233](#).



NOTE: The confirmed-commit operation is not available for a private copy. For information about using that operation for the regular candidate configuration, see [“Committing the Candidate Configuration Only After Confirmation Using the Junos XML Protocol” on page 223](#).

Related Documentation

- [Committing the Candidate Configuration Using the Junos XML Protocol on page 218](#)

- [Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol on page 225](#)
- [<commit-configuration> on page 104](#)

Committing a Configuration at a Specified Time Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to commit a configuration at a specified time in the future, a client application encloses the **<at-time>** element in a **<commit-configuration>** and an **<rpc>** element:

```
<rpc>
  <commit-configuration>
    <at-time>time</at-time>
  </commit-configuration>
</rpc>
```

To indicate when to perform the commit operation, the application includes one of three types of values in the **<at-time>** tag element:

- The string **reboot**, to commit the configuration the next time the device reboots.
- A time value of the form *hh:mm[:ss]* (hours, minutes, and optionally seconds), to commit the configuration at the specified time, which must be after the time at which the application emits the **<commit-configuration>** tag element, but before 11:59:59 PM on the current day. For example, if the **<at-time>** tag element encloses the value 02:00 (2:00 AM) and the application emits the **<commit-configuration>** tag element at 2:10 AM, the commit will never take place, because the scheduled time has already passed for that day.

Use 24-hour time; for example, 04:30:00 means 4:30:00 AM and 20:00 means 8:00 PM. The time is interpreted relative to the clock and time zone settings on the device..

- A date and time value of the form *yyyy-mm-dd hh:mm[:ss]* (year, month, date, hours, minutes, and optionally seconds), to commit the configuration at the specified day and time, which must be after the **<commit-configuration>** tag element is emitted.

Use 24-hour time; for example, 2006-08-21 15:30:00 means 3:30 PM on August 21, 2006. The time is interpreted relative to the clock and time zone settings on the device.



NOTE: The specified time must be more than 1 minute later than the current time on the device.

The Junos XML protocol server immediately checks the configuration for syntactic correctness and returns `<rpc-reply>`, `<commit-results>`, and `<routing-engine>` tag elements. If the syntax check succeeds, the `<routing-engine>` tag element encloses the `<commit-check-success/>` tag and the `<name>` tag element, which reports the name of the Routing Engine on which the check succeeded (re0 on devices that use a single Routing Engine, and either re0 or re1 on devices that can have two Routing Engines). It also encloses an `<output>` tag element that reports the time at which the commit will occur:

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>(re0 | re1)</name>
      <commit-check-success/>
      <output>commit at will be executed at timestamp</output>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

The configuration is scheduled for commit at the specified time. The Junos XML protocol server does not emit additional tag elements when it performs the actual commit operation.

If the configuration is not syntactically correct, an `<xnm:error>` tag element encloses tag elements that describe the error. The commit operation is not scheduled.

The `<at-time>` tag element can be combined with the `<synchronize/>` tag, the `<log/>` tag element, or both. For more information, see [“Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol” on page 225](#) and [“Logging a Message About a Commit Operation Using the Junos XML Protocol” on page 233](#).

The following example shows how to schedule a commit operation for 10:00 PM on the current day.

Client Application Junos XML Protocol Server

```
<rpc>
  <commit-configuration>
    <at-time>22:00</at-time>
  </commit-configuration>
</rpc>

<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re1</name>
      <commit-check-success/>
      <output>commit at will be executed at date 22:00:00 timezone</output>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

T1182

Related Documentation

- [Committing the Candidate Configuration Using the Junos XML Protocol on page 218](#)
- [Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol on page 225](#)
- [<commit-configuration> on page 104](#)

Committing the Candidate Configuration Only After Confirmation Using the Junos XML Protocol

When you commit the candidate configuration on a device running Junos OS, it becomes the active configuration on the routing, switching, or security platform. For more detailed information about commit operations, including a discussion of the interaction among different variants of the operation, see the *CLI User Guide*.

When you commit the candidate configuration, you can require an explicit confirmation for the commit to become permanent. The confirmed commit operation is useful for verifying that a configuration change works correctly and does not prevent management access to the device. If the change prevents access or causes other errors, the automatic rollback to the previous configuration restores access after the rollback deadline passes. If the commit is not confirmed within the specified amount of time, which is 600 seconds [10 minutes] by default, the device automatically retrieves and commits (rolls back to) the previously committed configuration.

In a Junos XML protocol session with a device running Junos OS, to commit the candidate configuration but require an explicit confirmation for the commit to become permanent, a client application encloses the empty `<confirmed/>` tag in the `<commit-configuration>` and `<rpc>` tag elements:

```
<rpc>
  <commit-configuration>
    <confirmed/>
  </commit-configuration>
</rpc>
```

To specify a different number of minutes for the rollback deadline, the application encloses a positive integer value in the `<confirm-timeout>` tag element:

```
<rpc>
  <commit-configuration>
    <confirmed/>
    <confirm-timeout>rollback-delay</confirm-timeout>
  </commit-configuration>
</rpc>
```



NOTE: You cannot perform a confirmed commit operation on a private copy of the configuration or on an instance of the ephemeral configuration database.

The Junos XML protocol server confirms that it committed the candidate configuration temporarily by returning the `<rpc-reply>`, `<commit-results>`, `<output>`, and `<routing-engine>` tag elements. If the initial commit operation succeeds, the `<routing-engine>` element encloses the `<commit-success/>` tag and the `<name>` tag element, which reports the name of the Routing Engine on which the commit operation succeeded (re0 on devices that use a single Routing Engine, and either re0 or re1 on devices that can have two Routing Engines):

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
```

```
        <output>commit confirmed will be automatically rolled back in 10
minutes unless confirmed</output>
        <routing-engine>
            <name>(re0 | re1)</name>
            <commit-success/>
        </routing-engine>
    </commit-results>
</rpc-reply>
```

If the Junos XML protocol server cannot commit the candidate configuration, the **<rpc-reply>** element instead encloses an **<xnm:error>** element explaining the reason for the failure. The most common causes are semantic or syntactic errors in the candidate configuration.

To delay the rollback to a time later than the current rollback deadline, the application emits the **<confirmed/>** tag in a **<commit-configuration>** tag element again before the deadline passes. Optionally, it can include the **<confirm-timeout>** element to specify how long to delay the next rollback; omit that tag element to delay the rollback by the default of 10 minutes. The client application can delay the rollback indefinitely by emitting the **<confirmed/>** tag repeatedly in this way.

To commit the configuration permanently, the client application emits one of the following tag sequences before the rollback deadline passes:

- The empty **<commit-configuration/>** tag enclosed in an **<rpc>** tag element. The rollback is canceled and the candidate configuration is committed immediately, as described in [“Committing the Candidate Configuration Using the Junos XML Protocol” on page 218](#). If the candidate configuration is still the same as the temporarily committed configuration, this effectively recommits the temporarily committed configuration:

```
<rpc>
  <commit-configuration/>
</rpc>
```

- The **<synchronize/>** tag enclosed in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
  </commit-configuration>
</rpc>
```

The rollback is canceled and the candidate configuration is checked and committed immediately on both Routing Engines, as described in [“Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol” on page 225](#). If a confirmed commit operation has been performed on both Routing Engines, then emitting the **<synchronize/>** tag cancels the rollback on both.

- The **<at-time>** tag element enclosed in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <at-time>time</at-time>
  </commit-configuration>
</rpc>
```


The rollback is canceled and the configuration is checked immediately for syntactic correctness, then committed at the scheduled time, as described in [“Committing a Configuration at a Specified Time Using the Junos XML Protocol” on page 221](#).

The `<confirmed/>` and `<confirm-timeout>` tag elements can be combined with the `<synchronize/>` tag, the `<log/>` tag element, or both. For more information, see [“Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol” on page 225](#) and [“Logging a Message About a Commit Operation Using the Junos XML Protocol” on page 233](#).

If another application uses the `<kill-session/>` tag element to terminate this application's session while a confirmed commit is pending (this application has committed changes but not yet confirmed them), the Junos XML protocol server that is servicing this session restores the configuration to its state before the confirmed commit instruction was issued. For more information about session termination, see [“Terminating Junos XML Protocol Sessions” on page 89](#).

The following example shows how to commit the candidate configuration on Routing Engine 1 with a rollback deadline of 20 minutes.

Client Application

```
<rpc>
  <commit-configuration>
    <confirmed/>
    <confirm-timeout>20</confirm-timeout>
  </commit-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re1</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

T1152

Related Documentation

- [Committing the Candidate Configuration Using the Junos XML Protocol on page 218](#)
- [<commit-configuration> on page 104](#)

Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol

A Routing Engine resides within a control plane. For single-chassis configurations, there is one control plane. In redundant systems, there are two control planes, the master plane and the backup plane. In multichassis configurations, the control plane includes all Routing Engines with the same Routing Engine designation. For example, all master Routing Engines reside within the *master* control plane, and all backup Routing Engines reside within the *backup* control plane.

Committing a configuration applies a new configuration to the device Engine. In a multichassis configuration, once a change to the configuration has been committed to the system, this change is propagated throughout the control plane using the distribution function.

In a redundant architecture, you can issue the **synchronize** command to commit the new configuration to both the master and the backup control planes. When issued, this command saves the current configuration to both device Routing Engines and commits the new configuration to both control planes. On a multichassis system, once the configuration has been committed on both planes, the distribution function distributes the new configuration across both planes. For more information about Routing Engine redundancy, see the *Junos OS High Availability Library for Routing Devices*.



NOTE: In a multichassis architecture with redundant control planes, there is a difference between synchronizing the two planes and distributing the configuration throughout each plane. Synchronization only occurs between the Routing Engines within the same chassis. Once this synchronization is complete, the new configuration is distributed to all other Routing Engines within the control planes of other chassis as a separate distribution function.

Because synchronization happens across two separate control planes, synchronizing configurations is only valid on redundant Routing Engine architectures. Further, re0 and re1 configuration groups must be defined on each routing, switching, or security platform. For more information about configuration groups, see the *CLI User Guide*.



NOTE: If you issue the **synchronize** command on a nonredundant Routing Engine system, the Junos XML protocol server commits the configuration on the one control plane.

For more information about synchronizing configurations, see the following sections:

- [Synchronizing the Candidate Configuration on Both Routing Engines on page 226](#)
- [Synchronizing the Ephemeral Configuration on Both Routing Engines on page 228](#)
- [Forcing a Synchronized Commit Operation on page 230](#)
- [Synchronizing the Candidate Configuration Simultaneously with Other Operations on page 231](#)

Synchronizing the Candidate Configuration on Both Routing Engines

To synchronize the candidate configuration or private copy on a redundant Routing Engine system, a client application encloses the empty **<synchronize/>** tag in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
  </commit-configuration>
</rpc>
```

The Junos XML protocol server verifies the configuration's syntactic correctness on the Routing Engine where the **<synchronize/>** tag is emitted (referred to as the local Routing Engine), copies the configuration to the remote Routing Engine and verifies its syntactic correctness there, and then commits the configuration on both Routing Engines.

The Junos XML protocol server encloses its response in **<rpc-reply>** and **<commit-results>** tag elements. It emits a separate **<routing-engine>** tag element for each operation on each Routing Engine:

- If the syntax check succeeds on a Routing Engine, the **<routing-engine>** tag element encloses the **<commit-check-success/>** tag and the **<name>** tag element, which reports the name of the Routing Engine on which the check succeeded (re0 or re1):

```
<routing-engine>
  <name>(re0 | re1)</name>
  <commit-check-success/>
</routing-engine>
```

If the configuration is incorrect, an **<xnm:error>** tag element encloses a description of the error.

- If the commit operation succeeds on a Routing Engine, the **<routing-engine>** tag element encloses the **<commit-success/>** tag and the **<name>** tag element, which reports the name of the Routing Engine on which the commit operation succeeded:

```
<routing-engine>
  <name>(re0 | re1)</name>
  <commit-success/>
</routing-engine>
```

If the commit operation fails, an **<xnm:error>** tag element encloses a description of the error. The most common causes of failure are semantic or syntactic errors in the configuration.

The following example shows how to commit and synchronize the candidate configuration on both Routing Engines.

Client Application

```
<rpc>
  <commit-configuration>
    <synchronize/>
  </commit-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re0</name>
      <commit-check-success/>
    </routing-engine>
    <routing-engine>
      <name>re1</name>
      <commit-check-success/>
    </routing-engine>
    <routing-engine>
      <name>re1</name>
      <commit-success/>
    </routing-engine>
    <routing-engine>
      <name>re0</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

T1153

Synchronizing the Ephemeral Configuration on Both Routing Engines

A client application can open and perform operations on an instance of the ephemeral configuration database on devices running Junos OS. After the application opens the ephemeral instance using the **<open-configuration>** operation with the appropriate child tags, the application can issue many of the same operations on the ephemeral instance as it would on the candidate configuration, including committing and synchronizing configuration data in that instance.

When dual Routing Engines are present, the device does not automatically synchronize ephemeral configuration data on the backup Routing Engine. To synchronize an open instance of the ephemeral configuration database to the other Routing Engine on a per-commit basis, a client application encloses the empty **<synchronize/>** tag in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
  </commit-configuration>
</rpc>
```

The Junos XML protocol server commits the configuration on the local Routing Engine and then copies the configuration to the remote Routing Engine and commits it. Unlike the standard commit model, the ephemeral commit model performs commit synchronize operations asynchronously. The requesting Routing Engine commits the ephemeral configuration and emits a commit complete notification without waiting for the other Routing Engine to first synchronize and commit the configuration.



NOTE: To synchronize an instance of the ephemeral configuration database to the other Routing Engine every time the instance is committed, the client must configure the `commit synchronize` statement at the `[edit system]` hierarchy level in the configuration for that ephemeral database instance.



NOTE: Multichassis and virtual chassis environments do not support synchronizing the ephemeral configuration database to the other Routing Engines.

The Junos XML protocol server reports the results of the commit operation for the local Routing Engine in `<rpc-reply>`, `<commit-results>`, and `<routing-engine>` tag elements. If the commit operation succeeds, the `<routing-engine>` tag element encloses the `<commit-success/>` tag and the `<name>` tag element, which reports the name of the Routing Engine on which the commit operation succeeded.

In addition, the Junos XML protocol server returns the `<commit-synchronize-server-success>` tag element enclosing the information for the synchronize operation, which includes the estimated time in seconds required for the synchronize operation to complete.

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re0</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
  <commit-synchronize-server-success>
    <current-job-id>0</current-job-id>
    <number-of-jobs>1</number-of-jobs>
    <estimated-time>60</estimated-time>
  </commit-synchronize-server-success>
</rpc-reply>
```

The `<commit-synchronize-server-success>` element indicates that the synchronize operation has been scheduled. Junos OS records the `UI_EPHEMERAL_COMMIT_SYNCHRONIZE_SUCCESS` or the `UI_EPHEMERAL_COMMIT_SYNCHRONIZE_FAILURE` event in the system log file to record the success or failure of the synchronize operation.



NOTE: Do *not* use the ephemeral database on devices running Junos OS that have nonstop active routing (NSR) enabled. Also, we do *not* recommend using the ephemeral database on devices that have GRES enabled.

Graceful Routing Engine switchover (GRES) enables a routing platform with redundant Routing Engines to continue forwarding packets, even if one Routing Engine fails. GRES requires that the configuration and certain state information are synchronized between the master and backup Routing Engines before a switchover occurs. Because of the synchronization requirements, we do *not* recommend using the ephemeral database on

devices that have GRES enabled, because in certain circumstances, the ephemeral database might not be synchronized between the master and backup Routing Engines.

When GRES is enabled on devices running Junos OS, the ephemeral configuration is not automatically synchronized to the backup Routing Engine when a commit synchronize operation is requested. If you elect to use the ephemeral database when GRES is enabled, you must explicitly configure the **allow-commit-synchronize-with-gres** statement at the **[edit system configuration-database ephemeral]** hierarchy level in the static configuration database to enable the device to synchronize ephemeral configuration data when a commit synchronize operation is requested. If GRES is enabled, and you do not configure the **allow-commit-synchronize-with-gres** statement, the device does not synchronize ephemeral configuration data under any circumstance.



NOTE: The device does not synchronize the ephemeral configuration database to the other Routing Engine when you issue the **commit synchronize** command in the CLI or when the **commit synchronize** statement is configured at the **[edit system]** hierarchy level in the static configuration database.

Forcing a Synchronized Commit Operation

The synchronize operation fails if the second Routing Engine's candidate configuration is locked. If a synchronization failure occurs, it is best to determine the cause of the failure, take corrective action, and then synchronize the two Routing Engines again. However, when necessary, you can use the **<force-synchronize/>** command to override a locked configuration and force the synchronization.



NOTE: When you use a **force-synchronize** command, any uncommitted changes to the configuration will be lost.

To force a synchronization, enclose the empty **<synchronize/>** and **<force-synchronize/>** tags in the **<rpc>** and **<commit-configuration>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <force-synchronize/>
  </commit-configuration>
</rpc>
```



NOTE: In a multichassis environment, synchronization occurs between Routing Engines on the same chassis. Once the synchronization occurs, the configuration changes are propagated across each control plane using the distribution function. If one or more Routing Engines are locked during the distribution of the configuration, the distribution and thus the synchronization will fail. You will need to clear the error in the remote chassis and run the **synchronize** command again.

The following example shows how to force a synchronization across both Routing Engine planes:

Client Application Junos XML Protocol Server

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <force-synchronize/>
  </commit-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos=
  "http://xml.juniper.net/junos/9.010/junos">
  <commit-results>
    <routing-engine junos:style="show-name">
      <name>re0</name>
      <commit-check-success/>
    </routing-engine>
    <routing-engine junos:style="show-name">
      <name>re1</name>
      <commit-success/>
    </routing-engine>
    <routing-engine junos:style="show-name">
      <name>re0</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

Synchronizing the Candidate Configuration Simultaneously with Other Operations

The **<synchronize/>** tag can be combined with the other tag elements that can occur within the **<commit-configuration>** tag element. The Junos XML protocol server checks, copies, and commits the configuration, and emits the same response tag elements as when the **<synchronize/>** tag is used by itself. The possible combinations are described in the following sections.

Verifying the Configuration on Both Routing Engines

To check the syntactic correctness of a local configuration on both Routing Engines without committing it, the application encloses the **<synchronize/>** and **<check/>** tag elements in **<commit-configuration>** and **<rpc>** tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <check/>
  </commit-configuration>
</rpc>
```

The **<force-synchronize/>** tag cannot be combined with the **<check/>** tag elements.

For more information about verifying configurations, see [“Verifying Configuration Syntax Using the Junos XML Protocol” on page 217](#).

**Scheduling
Synchronization for a
Specified Time**

To commit a configuration on both Routing Engines at a specified time in the future, the application encloses the `<synchronize/>` and `<at-time>` tag elements in `<commit-configuration>` and `<rpc>` tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <at-time>time</at-time>
  </commit-configuration>
</rpc>

<rpc>
  <commit-configuration>
    <force-synchronize/>
    <at-time>time</at-time>
  </commit-configuration>
</rpc>
```

As when the `<at-time>` tag element is emitted by itself, the Junos XML protocol server verifies syntactic correctness immediately and does not emit additional tag elements when it actually performs the commit operation on each Routing Engine.

**Synchronizing
Configurations but
Requiring Confirmation**

To commit the candidate configuration on both Routing Engines but require confirmation for the commit to become permanent, the application encloses the `<synchronize/>`, `<confirmed/>`, and (optionally) `<confirm-timeout>` tag elements in `<commit-configuration>` and `<rpc>` tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <confirmed/>
    [<confirm-timeout>minutes</confirm-timeout>]
  </commit-configuration>
</rpc>
```

The same rollback deadline applies to both Routing Engines and can be extended on both at once by again emitting the `<synchronize/>`, `<confirmed/>`, and (optionally) `<confirm-timeout>` tag elements on the Routing Engine where the tag elements were emitted the first time.

The `<force-synchronize/>` tag cannot be combined with the `<confirmed/>` and `<confirm-timeout>` tag elements.

For more information about confirmed commit operations, see [“Committing the Candidate Configuration Only After Confirmation Using the Junos XML Protocol” on page 223](#).

**Logging a Message
About Synchronized
Configurations**

To synchronize configurations and record a log message when the commit succeeds on each Routing Engine, the application encloses the `<synchronize/>` and `<log/>` tag elements in `<commit-configuration>` and `<rpc>` tag elements:

```
<rpc>
  <commit-configuration>
    <synchronize/>
    <log>message</log>
  </commit-configuration>
</rpc>
<rpc>
  <commit-configuration>
```



```

    <force-synchronize/>
    <log>message</log>
  </commit-configuration>
</rpc>

```

The commit operation proceeds as previously described in the `<synchronize/>` or `<force-synchronize/>` tag descriptions. The message for each Routing Engine is recorded in the commit history log maintained by that Routing Engine. For more information about logging, see “Logging a Message About a Commit Operation Using the Junos XML Protocol” on page 233.

- Related Documentation**
- [Committing the Candidate Configuration Using the Junos XML Protocol on page 218](#)
 - [Logging a Message About a Commit Operation Using the Junos XML Protocol on page 233](#)
 - [<commit-configuration> on page 104](#)

Logging a Message About a Commit Operation Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to log a comment when performing a commit operation, a client application encloses the `<log>` tag element in `<commit-configuration>` and `<rpc>` tag elements:

```

<rpc>
  <commit-configuration>
    <log>message</log>
  </commit-configuration>
</rpc>

```

The `<log>` element can be combined with other tags within the `<commit-configuration>` tag element (the `<at-time>`, `<confirmed/>`, and `<confirm-timeout>`, or `<synchronize/>` tag elements) and does not change the effect of the operation. When the `<log>` tag element is emitted by itself, the associated commit operation begins immediately.

The following example shows how to log a message as the candidate configuration is committed.

Client Application

```

<rpc>
  <commit-configuration>
    <log>Enable xnm-ssl service</log>
  </commit-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re0</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>

```

T1154

The commit history, which includes any commit comments, stores an entry for each pending commit and up to 50 previous commits for the standard configuration database. To request the history, a client application encloses the `<get-commit-information/>` tag in `<rpc>` tag elements. The equivalent operational mode CLI command is `show system commit`.

```
<rpc>
  <get-commit-information/>
</rpc>
```

The Junos XML protocol server encloses the information in `<commit-information>` and `<rpc-reply>` tag elements. For information about the child tag elements of the `<commit-information>` tag element, see its entry in the *Junos XML API Operational Developer Reference*.

```
<rpc-reply xmlns:junos="URL">
  <commit-information>
    <!-- tag elements representing the commit log -->
  </commit-information>
</rpc-reply>
```

The following example shows how to request the commit log.

Client Application Junos XML Protocol Server

```
<rpc>
  <get-commit-information/>
</rpc>

<rpc-reply xmlns:junos="URL">
  <commit-information>
    <commit-history>
      <sequence-number>0</sequence-number>
      <user>barbara</user>
      <client>other</client>
      <date-time junos:seconds="1058370173">2003-07-16 08:42:53 PDT</date-time>
      <log>Enable xnm-ssl service</log>
    </commit-history>
    <commit-history>
      <sequence-number>1</sequence-number>
      <user>root</user>
      <client>other</client>
      <date-time junos:seconds="1058322166">2003-07-15 19:22:46 PDT</date-time>
    </commit-history>
    <commit-history>
      <sequence-number>2</sequence-number>
      <user>root</user>
      <client>cli</client>
      <date-time junos:seconds="1058219717">2003-07-14 14:55:17 PDT</date-time>
    </commit-history>
    .
    .
    .
  </commit-information>
</rpc-reply>
```

T1183

Related Documentation

- [Committing the Candidate Configuration Using the Junos XML Protocol on page 218](#)
- [Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol on page 225](#)
- [<commit-configuration> on page 104](#)

Viewing the Configuration Revision Identifier for Determining Synchronization Status of Devices with NMS

Out-of-band configuration changes are configuration changes made to a device outside of the network management server (NMS) application, such as Junos Space. For example, configuration changes can be performed on a device using the device CLI, using the device Web-based management interface (the J-Web interface or Web View), or using the Junos Space Network Management Platform configuration editor. As a result, there is a requirement for a configuration revision identifier to determine whether the configuration settings on devices being managed by an NMS application are in sync with the CLI of devices running Junos OS.

A configuration revision identifier might not be necessary if the NMS application is the only utility that is used to modify the configuration of a device. However, in a real-world network deployment, out-of-band configuration commits might occur on a device, such as during a maintenance window for support operations. In such cases, the NMS application might not detect these out-of-band commits. To solve this problem, starting in Junos OS Release 16.1, a configuration revision identifier tag, **<commit-revision-information>**, is supported within the **<commit-results>** tag. The configuration revision identifier is a string (for example, re0-1365168149-1), which has the following format:

```
<routing-engine-name>-<timestamp>-<counter>
```

Different platforms contain different Routing Engine names as follows:

- Dual Routing Engines of MX Series routers—re0, re1
- SRX Series Chassis Cluster—node0, node1, node2, and so on
- MX Series Virtual Chassis—member0-re0, member0-re1, member1-re0, and so on
- EX Series Virtual Chassis—fpc0, fpc1, and so on

The Routing Engine name is different from the user-configured hostname of the device. The Routing Engine name is used to identify the source device that has performed the configuration change on a device. When the revision number is computed, the time is displayed in the Unix epoch format (also known as Unix time or POSIX time). The counter increments by one for every commit operation performed. The NMS application considers the configuration revision identifier as an entire string and does not parse individual substrings of this revision identifier.

On every successful commit, in addition to the commit-success message, Junos OS also returns the old revision number and updated revision number. The NMS application can store this revision locally. The NMS application can query Junos OS to retrieve the latest revision number and compare it against the revision number stored locally to validate whether it is out-of-sync or in-sync with the device. The NETCONF remote procedure calls (RPCs) and the CLI (**show system commit revision detail**) are used to query Junos OS for the revision string.

The following is an example of the RPC reply that contains the commit revision details with the new **<commit-revision-information>** tag included:

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
  <commit-results>
    <routing-engine>
      <name>re0</name>
      <commit-success/>
      <commit-revision-information>
        <old-db-revision>re0-1446493106-63</old-db-revision>
        <new-db-revision>re0-1446493220-64</new-db-revision>
      </commit-revision-information>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

**Related
Documentation**

- [Committing the Candidate Configuration Using the Junos XML Protocol on page 218](#)
- [Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol on page 225](#)
- [<commit-revision-information> on page 128](#)
- [<commit-configuration> on page 104](#)

CHAPTER 13

Using the Ephemeral Configuration Database

- [Understanding the Ephemeral Configuration Database on page 237](#)
- [Unsupported Configuration Statements in the Ephemeral Configuration Database on page 242](#)
- [Enabling and Configuring Instances of the Ephemeral Configuration Database on page 244](#)
- [Committing an Instance of the Ephemeral Configuration Database Using the NETCONF or Junos XML Protocol on page 249](#)

Understanding the Ephemeral Configuration Database

The *ephemeral database* is an alternate configuration database that provides a fast programmatic interface for performing configuration updates on devices running Junos OS. The ephemeral database enables Juniper Extension Toolkit (JET) applications and NETCONF and Junos XML management protocol client applications to concurrently load and commit configuration changes to a device running Junos OS and with significantly greater throughput than when committing data to the candidate configuration database.

The different aspects of the ephemeral configuration database are discussed in the following sections:

- [Ephemeral Configuration Database Overview on page 237](#)
- [Ephemeral Database Instances on page 239](#)
- [Ephemeral Database Commit Model on page 240](#)

Ephemeral Configuration Database Overview

When managing devices running Junos OS, the recommended and most common method to configure the device is to modify and commit the candidate configuration, which corresponds to a persistent (static) configuration database. The standard Junos OS commit operation handles configuration groups, macros, and commit scripts; performs commit checks to validate the configuration's syntax and semantics; and stores copies of the committed configurations. The standard commit model is robust, because it prevents configuration errors and enables you to roll back to a previously committed

configuration. However, in some cases, the commit operation can consume a significant amount of time and device resources.

JET applications and NETCONF and Junos XML protocol client applications can also configure the ephemeral database. The ephemeral database is an alternate configuration database that provides a configuration layer separate from both the candidate configuration database and the configuration layers of other client applications. The ephemeral commit model enables devices running Junos OS to commit and merge changes from multiple clients and execute the commits with significantly greater throughput than when committing data to the candidate configuration database. Thus, the ephemeral database is advantageous in dynamic environments where fast provisioning and rapid configuration changes are required, such as in large data centers.

A commit operation on the ephemeral database requires less time than the same operation on the static database, because the ephemeral database is not subject to the same verification required in the static database. As a result, the ephemeral commit model provides better performance than the standard commit model but at the expense of some of the more robust features present in the standard model. The ephemeral commit model has the following restrictions:

- Configuration data syntax is validated, but configuration data semantics are not validated.
- Certain configuration statements are not supported as described in [“Unsupported Configuration Statements in the Ephemeral Configuration Database” on page 242](#).
- Configuration groups and interface ranges are not processed.
- Macros, commit scripts, and translation scripts are not processed.
- Previous versions of the ephemeral configuration are not archived.
- Ephemeral configuration data does not persist across reboots.
- Ephemeral configuration data does not persist when installing a package that requires rebuilding the Junos OS schema, for example, an OpenConfig or YANG package.
- Ephemeral configuration data is not displayed in the normal configuration using standard show commands.



NOTE: We strongly recommend that you exercise caution when using the ephemeral configuration database, because committing invalid configuration data can corrupt the ephemeral database and result in disruption to the system or network.

Junos OS validates the syntax but not the semantics, or constraints, of the configuration data committed to the ephemeral database. For example, if the configuration references an undefined routing policy, the configuration might be syntactically correct, but it would be semantically incorrect. The standard commit model generates a commit error in this case, but the ephemeral commit model does not. Therefore, it is imperative to validate all configuration data before committing it to the ephemeral database. If you commit configuration data that is invalid or results in undesirable network disruption, you must

delete the problematic data from the database, or if necessary, reboot the device, which deletes all ephemeral configuration data.



NOTE: The ephemeral configuration database stores internal version information in addition to configuration data. As a result, the size of the ephemeral configuration database is always larger than the static configuration database for the same configuration data, and most operations on the ephemeral database, whether additions, modifications, or deletions, increase the size of the database.



NOTE: If the ephemeral configuration database is present on a device running Junos OS, commit operations on the static configuration database might take longer, because additional operations must be performed to merge the static and ephemeral configuration data.

Ephemeral Database Instances

Devices running Junos OS provide a default ephemeral database instance, which is enabled by default, as well as the ability to define up to eight user-defined instances of the ephemeral configuration database. JET applications and NETCONF and Junos XML protocol client applications can concurrently load and commit data to separate instances of the ephemeral database. The active device configuration is a merged view of the static and ephemeral configuration databases.

Ephemeral database instances are useful in scenarios where multiple client applications might need to simultaneously update a device configuration, such as when two or more SDN controllers simultaneously push configuration data to the same device. In the standard commit model, one controller might have an exclusive lock on the candidate configuration, thereby preventing the other controller from modifying it. By using separate ephemeral instances, the controllers can deploy the changes at the same time.



NOTE: Although applications can simultaneously load and commit data to different instances of the ephemeral database, commits issued at the same time from different ephemeral instances are queued and processed serially by the device.

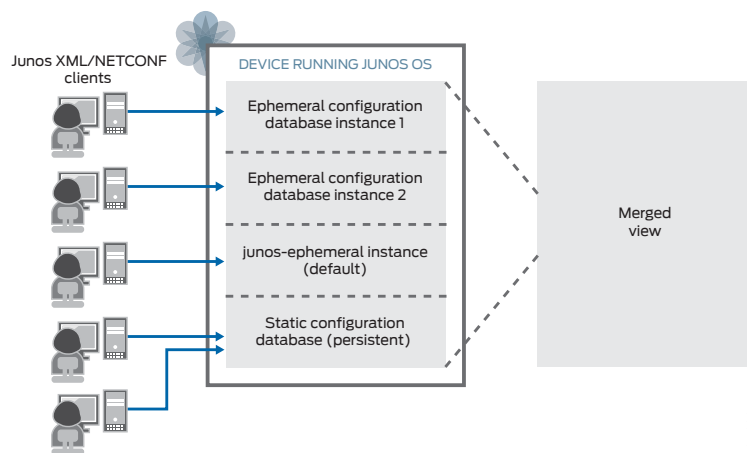
The Junos OS processes read the configuration data from both the static configuration database and the ephemeral configuration database. When one or more ephemeral database instances are in use and there is conflicting data, statements in a database with a higher priority override those in a database with a lower priority. A user-defined instance of the ephemeral configuration database has higher priority than the default ephemeral database instance, which has higher priority than the static configuration database. If there are multiple user-defined ephemeral instances, the priority is determined by the order in which the instances are listed in the configuration at the `[edit system configuration-database ephemeral]` hierarchy level, running from highest to lowest priority.

Consider the following configuration:

```
system {
  configuration-database {
    ephemeral {
      instance 1;
      instance 2;
    }
  }
}
```

Figure 1 on page 240 illustrates the order of priority of the ephemeral database instances and the static (committed) configuration database. In this example, ephemeral database instance 1 has the highest priority, followed by ephemeral database instance 2, then the default ephemeral database instance, and finally the static configuration database.

Figure 1: Ephemeral Database Instances



Ephemeral Database Commit Model

JET applications and NETCONF and Junos XML protocol client applications can modify the ephemeral configuration database. JET applications must send configuration requests as pairs of load and commit operations. NETCONF and Junos XML protocol client applications can perform multiple load operations before executing a commit operation.

The ephemeral database only supports **load merge** and **load set** operations, but configuration data can be uploaded in any of the supported formats including Junos XML elements, formatted ASCII text, **set** commands, or JavaScript Object Notation (JSON). By default, if a client disconnects from a session, Junos OS discards any uncommitted configuration changes in the ephemeral instance, but configuration data that has already been committed to the ephemeral instance by that client is unaffected.



NOTE: In the ephemeral commit model, Junos OS validates the syntax but not the semantics, or constraints, of the configuration data committed to the ephemeral database. Therefore, it is imperative to validate all configuration data before loading it into the ephemeral database and committing it on the device.

During the commit operation, Junos OS validates the syntax, but not the semantics, of the configuration data committed to the ephemeral database. When the commit is complete, Junos OS notifies the affected system processes, which read the updated configuration and merge the ephemeral data into the active configuration according to the rules of prioritization described in [“Ephemeral Database Instances” on page 239](#). The active device configuration is a merged view of the static and ephemeral configuration databases.

When dual Routing Engines are present, the ephemeral commit model does not automatically synchronize ephemeral configuration data on the backup Routing Engine. Client applications can elect to synchronize the data in an ephemeral instance on a per-commit basis or every time the instance is committed. Unlike the standard commit model, the ephemeral commit model performs commit synchronize operations asynchronously. The requesting Routing Engine commits the ephemeral configuration and emits a commit complete notification without waiting for the other Routing Engine to first synchronize and commit the configuration.



NOTE: Multichassis and virtual chassis environments do not support synchronizing the ephemeral configuration database to the other Routing Engines.



NOTE: Do *not* use the ephemeral database on devices running Junos OS that have nonstop active routing (NSR) enabled. Also, we do *not* recommend using the ephemeral database on devices that have graceful Routing Engine switchover (GRES) enabled.

Graceful Routing Engine switchover (GRES) enables a routing platform with redundant Routing Engines to continue forwarding packets, even if one Routing Engine fails. GRES requires that the configuration and certain state information are synchronized between the master and backup Routing Engines before a switchover occurs. Because of the synchronization requirements, we do *not* recommend using the ephemeral database on devices that have GRES enabled, because in certain circumstances, the ephemeral database might not be synchronized between the master and backup Routing Engines.

For example, because commit synchronize operations on the ephemeral database are asynchronous, the backup and master Routing Engines might not synchronize if the operation is interrupted by a sudden failover or power outage. Furthermore, if the backup Routing Engine restarts, its ephemeral configuration data is deleted, because ephemeral data does not persist across reboots. When GRES is enabled and the backup Routing

Engine restarts for any reason, its static configuration database is automatically synchronized with the database on the master Routing Engine, but the ephemeral configuration database is *not* synchronized.

When GRES is enabled on devices running Junos OS, the ephemeral configuration is not automatically synchronized to the backup Routing Engine when a commit synchronize operation is requested. If you elect to use the ephemeral database with the aforementioned caveats when GRES is enabled, you must explicitly configure the **allow-commit-synchronize-with-gres** statement at the **[edit system configuration-database ephemeral]** hierarchy level in the static configuration database to enable the device to synchronize ephemeral configuration data when a commit synchronize operation is requested. If GRES is enabled, and you do not configure the **allow-commit-synchronize-with-gres** statement, the device does not synchronize ephemeral configuration data under any circumstance.

For detailed information about committing and synchronizing instances of the ephemeral configuration database, see the following topics:

- [Committing an Instance of the Ephemeral Configuration Database Using the NETCONF or Junos XML Protocol on page 249](#)
- [Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol on page 225](#)

Related Documentation


- [Enabling and Configuring Instances of the Ephemeral Configuration Database on page 244](#)
- *Example: Configuring the Ephemeral Configuration Database Using NETCONF*

Unsupported Configuration Statements in the Ephemeral Configuration Database

The ephemeral database is an alternate configuration database that enables Juniper Extension Toolkit (JET) applications and NETCONF and Junos XML protocol client applications to simultaneously load and commit configuration changes on devices running Junos OS and with significantly greater throughput than when committing data to the candidate configuration database. To improve commit performance, the ephemeral commit process does not perform all of the operations and validations executed by the standard commit model. As a result, there are some features that cannot be configured through the ephemeral database. For example, the ephemeral configuration database does not support configuring interface alias names or any type of Spanning Tree Protocol (xSTP, where the “x” represents the STP type).

The following configuration statements are not supported in the ephemeral configuration database. If a client attempts to configure an unsupported statement in an ephemeral instance, the server returns an error during the load operation. The configuration statements are grouped under their top-level configuration statement.

[edit]	[edit apply-groups]
	[edit access]
	[edit chassis]

	[edit dynamic-profiles] [edit security] (SRX Series only)
[edit interfaces]	[edit interfaces <i>interface-name</i> unit <i>logical-unit-number</i> alias <i>alias-name</i>]
[edit logical-systems]	[edit logical-systems <i>logical-system-name</i> interfaces <i>interface-name</i> unit <i>logical-unit-number</i> alias <i>alias-name</i>] [edit logical-systems <i>logical-system-name</i> policy-options prefix-list <i>name</i> apply-path <i>path</i>] [edit logical-systems <i>logical-system-name</i> protocols mstp] [edit logical-systems <i>logical-system-name</i> protocols rstp] [edit logical-systems <i>logical-system-name</i> protocols vstp] [edit logical-systems <i>logical-system-name</i> system processes routing]
[edit policy-options]	[edit policy-options prefix-list <i>name</i> apply-path <i>path</i>]
[edit protocols]	[edit protocols mstp] [edit protocols rstp] [edit protocols vstp]
[edit routing-instances]	[edit routing-instances <i>instance-name</i> protocols mstp] [edit routing-instances <i>instance-name</i> protocols rstp] [edit routing-instances <i>instance-name</i> protocols vstp]
[edit security]	[edit security group-vpn member ipsec vpn] [edit security ssh-known-hosts host <i>hostname</i>]
<div style="display: flex; align-items: center;">  <div> <p>NOTE: The ephemeral configuration database does not support configuring the [edit security] hierarchy on SRX Series Services Gateways.</p> </div> </div>	
[edit services]	[edit services ssl initiation profile] [edit services ssl proxy profile] [edit services ssl termination profile]
[edit system]	[edit system archival] [edit system commit delta-export] [edit system commit fast-synchronize] [edit system commit notification] [edit system commit peers] [edit system commit peers-synchronize] [edit system commit persist-groups-inheritance] [edit system commit server] [edit system compress-configuration-files] [edit system configuration-database] [edit system extensions] [edit system fips] [edit system host-name] [edit system license]

```
[edit system login]
[edit system master-password]
[edit system max-configurations-on-flash]
[edit system radius-options]
[edit system regex-additive-logic]
[edit system scripts]
[edit system services extension-service notification allow-clients address]
[edit system time-zone]
```

Related Documentation

- [Understanding the Ephemeral Configuration Database on page 237](#)

Enabling and Configuring Instances of the Ephemeral Configuration Database

The ephemeral database is an alternate configuration database that enables multiple client applications to concurrently load and commit configuration changes to a device running Junos OS and with significantly greater throughput than when committing data to the candidate configuration database. Junos OS provides a default ephemeral database instance as well as the ability to define up to eight user-defined instances of the ephemeral configuration database.



NOTE: When you configure statements at the `[edit system configuration-database ephemeral]` hierarchy level and commit the configuration, all Junos OS processes must check and evaluate their complete configuration, which might cause a spike in CPU utilization, potentially impacting other critical software processes.

NETCONF and Junos XML protocol client applications and JET applications can update the ephemeral configuration database. The following sections detail how to enable instances of the ephemeral configuration database on devices running Junos OS, configure the instances using NETCONF and Junos XML protocol operations, and display ephemeral configuration data in the Junos OS CLI. For information about using JET applications to configure the ephemeral configuration database, see the [JET documentation](#).

1. [Enabling Ephemeral Database Instances on page 245](#)
2. [Configuring Ephemeral Database Options on page 245](#)
3. [Opening Ephemeral Database Instances on page 246](#)
4. [Configuring Ephemeral Database Instances on page 246](#)
5. [Displaying Ephemeral Configuration Data in the Junos OS CLI on page 248](#)

Enabling Ephemeral Database Instances

The default ephemeral database instance is automatically enabled on devices running Junos OS. However, you must configure all user-defined instances of the ephemeral configuration database before using them.

Before enabling any instances of the ephemeral configuration database, ensure that the device meets the following requirements:

- Device is running Junos OS Release 16.2R2 or a later release.

To enable a user-defined instance of the ephemeral configuration database:

1. Configure the name of the instance.

```
[edit system configuration-database ephemeral]
user@host# set instance instance-name
```



NOTE: Junos OS determines the priority of ephemeral database instances by the order in which they are listed in the configuration. By default, newly configured instances are placed at the end of the list and have lower priority when resolving conflicting configuration statements. When you configure a new instance, you can specify its placement in the configuration by using the insert command instead of the set command.

2. Commit the configuration.

```
[edit system configuration-database ephemeral]
user@host# commit
```

Configuring Ephemeral Database Options

You can configure several options for the ephemeral configuration database, which are outlined in this section.

1. (Optional) To disable the default instance of the ephemeral configuration database, configure the **ignore-ephemeral-default** statement.

```
[edit system configuration-database ephemeral]
user@host# set ignore-ephemeral-default
```

2. (Optional) When graceful Routing Engine switchover (GRES) is enabled, configure the **allow-commit-synchronize-with-gres** statement to enable the device to synchronize ephemeral configuration data when a commit synchronize operation is requested.

```
[edit system configuration-database ephemeral]
user@host# set allow-commit-synchronize-with-gres
```

3. Commit the configuration.

```
[edit system configuration-database ephemeral]
```

```
user@host# commit
```

Opening Ephemeral Database Instances

A client application must open an ephemeral database instance before viewing or modifying it. Within a NETCONF or Junos XML protocol session, a client application opens the ephemeral database instance by using the Junos XML protocol **<open-configuration>** operation with the appropriate child tags. Opening the ephemeral instance automatically acquires an exclusive lock on it.



NOTE: To configure devices running Junos OS using NETCONF, the NETCONF-over-SSH service must be enabled on the device. NETCONF sessions must also include the closing sequence `]]>]]>` at the end of each remote procedure call (RPC) request.

- To open the default instance of the ephemeral database, a client application emits the **<open-configuration>** element and includes the **<ephemeral/>** child tag.

```
<rpc>
  <open-configuration>
    <ephemeral/>
  </open-configuration>
</rpc>
```

- To open a user-defined instance of the ephemeral database, a client application emits the **<open-configuration>** element and includes the **<ephemeral-instance>** element and the instance name.

```
<rpc>
  <open-configuration>
    <ephemeral-instance>instance-name</ephemeral-instance>
  </open-configuration>
</rpc>
```

Configuring Ephemeral Database Instances

Client applications update the ephemeral configuration database using NETCONF and Junos XML protocol operations. Only a subset of the operations' attributes and options are available for use when updating the ephemeral configuration database. For example, the ephemeral database only supports **load merge** and **load set** operations, and options and attributes that reference groups, interface ranges, or commit scripts, or that roll back the configuration cannot be used with the ephemeral database.

Client applications load and commit configuration data to an open instance of the ephemeral configuration database. Configuration data can be uploaded in any of the supported formats including Junos XML elements, formatted ASCII text, set commands, or JavaScript Object Notation (JSON). By default, if a client disconnects from a session or closes the ephemeral database instance before committing new changes, Junos OS discards any uncommitted data, but configuration data that has already been committed to the ephemeral database instance by that client is unaffected.

To update, commit, and close an open instance of the ephemeral configuration database, client applications perform the following tasks:

1. Load configuration data into the ephemeral database instance by performing one or more load operations.

Client applications emit the `<load-configuration>` operation in a Junos XML protocol session or the `<load-configuration>` or `<edit-config>` operation in a NETCONF session and include the appropriate attributes and tags for the data.

```
<rpc>
  <load-configuration action="(merge | set)" format="(text | json | xml)">
    <!--configuration-data-->
  </load-configuration>
</rpc>
```



NOTE: The only acceptable format for `action="set"` is `"text"`. For more information about the `<load-configuration>` operation, see [“<load-configuration>” on page 114](#).

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <!--configuration-data-->
  </edit-config>
</rpc>
```



NOTE: The target value `<candidate/>` can refer to either the open configuration database, or if there is no open database, to the candidate configuration. If a client application issues the Junos XML protocol `<open-configuration>` operation to open an ephemeral instance before executing the `<edit-config>` operation, Junos OS performs the `<edit-config>` operation on the open instance of the ephemeral configuration database. Otherwise, the operation is performed on the candidate configuration.



NOTE: Commit operations on the ephemeral configuration database generate system log messages, but load operations do not.

2. (Optional) Review the updated configuration in the open ephemeral instance by emitting the `<get-configuration/>` operation in a Junos XML protocol session or the `<get-configuration/>` or `<get-config>` operation in a NETCONF session.

```
<rpc>
  <get-configuration format="(json | set | text | xml)" />
</rpc>
```

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
  </get-config>
</rpc>
```

3. Commit the configuration changes by emitting the **<commit-configuration/>** operation in a Junos XML protocol session or the **<commit-configuration/>** or **<commit/>** operation in a NETCONF session.

On devices with dual Routing Engines, include the **<synchronize/>** tag in the **<commit-configuration>** element to synchronize the data to the other Routing Engine.

```
<rpc>
  <commit-configuration>
    <synchronize/>
  </commit-configuration>
</rpc>
```

```
<rpc>
  <commit/>
</rpc>
```



NOTE: To automatically synchronize the configuration data in an ephemeral database instance to the other Routing Engine for every commit operation, include the **commit synchronize** statement at the **[edit system]** hierarchy level within the configuration for the specific ephemeral instance.



NOTE: After a client application commits changes to the ephemeral database instance, Junos OS merges the ephemeral data into the active configuration according to the rules of prioritization.

4. Repeat steps 1 through 3 for any subsequent updates to the ephemeral database instance.
5. Close the ephemeral database instance, which releases the exclusive lock.

```
<rpc>
  <close-configuration/>
</rpc>
```

Displaying Ephemeral Configuration Data in the Junos OS CLI

The active device configuration is a merged view of the static and ephemeral configuration databases. However, when you display the configuration in the CLI using the **show configuration** command in operational mode, the output does not include ephemeral

configuration data. You can display the data in a specific instance of the ephemeral database or display a merged view of the static and ephemeral configuration databases in the CLI by using variations of the **show ephemeral-configuration** command.

To view the configuration data in the default instance of the ephemeral configuration database, issue the **show ephemeral-configuration** operational command in the CLI.

```
user@host> show ephemeral-configuration
```

To view the configuration data in a user-defined instance of the ephemeral configuration database, issue the **show ephemeral-configuration instance-name** operational command.

```
user@host> show ephemeral-configuration instance-name
```

To view the complete post-inheritance configuration merged with the configuration data in all instances of the ephemeral database, issue the **show ephemeral-configuration | display merge** operational command.

```
user@host> show ephemeral-configuration | display merge
```

Related Documentation

- *Example: Configuring the Ephemeral Configuration Database Using NETCONF*
- [Understanding the Ephemeral Configuration Database on page 237](#)
- [Committing an Instance of the Ephemeral Configuration Database Using the NETCONF or Junos XML Protocol on page 249](#)
- [ephemeral on page 382](#)
- [show ephemeral-configuration on page 388](#)

Committing an Instance of the Ephemeral Configuration Database Using the NETCONF or Junos XML Protocol

The ephemeral database is an alternate configuration database that enables NETCONF and Junos XML protocol client applications to simultaneously load and commit configuration changes on devices running Junos OS and with significantly greater throughput than when committing data to the candidate configuration database. Client applications can commit the configuration data in an open instance of the ephemeral configuration database so that it becomes part of the active configuration on the routing, switching, or security platform by using the **<commit-configuration/>** operation in a Junos XML protocol session or the **<commit-configuration/>** or **<commit/>** operation in a NETCONF session. When you commit ephemeral configuration data on a device running Junos OS, the device's active configuration is a merged view of the static and ephemeral configuration databases.



NOTE: In the ephemeral commit model, Junos OS validates the syntax but not the semantics, or constraints, of the configuration data committed to the ephemeral database. Therefore, it is imperative to validate all configuration data before loading it into the ephemeral database and committing it on the device.

In a Junos XML protocol session with a device running Junos OS, a client application commits the configuration data in an open instance of the ephemeral configuration database by enclosing the `<commit-configuration/>` tag in an `<rpc>` tag element (just as for the candidate configuration).

```
<rpc>
  <commit-configuration/>
</rpc>
```

The Junos XML protocol server reports the results of the commit operation in `<rpc-reply>`, `<commit-results>`, and `<routing-engine>` tag elements. If the commit operation succeeds, the `<routing-engine>` tag element encloses the `<commit-success/>` tag and the `<name>` tag element, which reports the name of the Routing Engine on which the commit operation succeeded.

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>routing-engine-name</name>
      <commit-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

In a NETCONF session with a device running Junos OS, a client application commits the configuration data in an open instance of the ephemeral configuration database by enclosing the `<commit/>` or `<commit-configuration/>` tag in an `<rpc>` tag element (just as for the candidate configuration).

```
<rpc>
  <commit/>
</rpc>
]]>]]>

<rpc>
  <commit-configuration/>
</rpc>
]]>]]>
```

The NETCONF server confirms that the commit operation was successful by returning the `<ok/>` tag in an `<rpc-reply>` tag element.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <ok/>
</rpc-reply>
]]>]]>
```

If the commit operation fails, the NETCONF server returns the `<rpc-reply>` element and `<rpc-error>` child element, which explains the reason for the failure.

The only variant of the commit operation supported for the ephemeral database is synchronizing the configuration on the other Routing Engine, as described in [“Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol”](#) on page 225.

After a client application commits an ephemeral instance, the data in that instance is merged into the ephemeral configuration database. The affected system processes parse the configuration and merge the ephemeral data with the data in the active

configuration. If there are conflicting statements in the static and ephemeral configuration databases, the data is merged according to specific rules of prioritization. Statements in a user-defined instance of the ephemeral configuration database have higher priority than statements in the default ephemeral database instance, which have higher priority than statements in the static configuration database. If there are multiple user-defined ephemeral instances, the priority is determined by the order in which the instances are configured at the `[edit system configuration-database ephemeral]` hierarchy level, running from highest to lowest priority.



NOTE: Although applications can simultaneously load and commit data to different instances of the ephemeral database, commits issued at the same time from different ephemeral instances are queued and processed serially by the device.



NOTE: If you commit ephemeral configuration data that is invalid or results in undesirable network disruption, you must delete the problematic data from the database, or if necessary, reboot the device, which deletes the configuration data in all instances of the ephemeral configuration database.

The active device configuration is a merged view of the static and ephemeral configuration databases. However, when you display the configuration in the CLI using the **show configuration** command in operational mode, the output does not include ephemeral configuration data. You can display the data in a specific instance of the ephemeral database or display a merged view of the static and ephemeral configuration databases in the CLI by using variations of the **show ephemeral-configuration** command.

Related Documentation

- [Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol on page 225](#)
- [Enabling and Configuring Instances of the Ephemeral Configuration Database on page 244](#)
- [Understanding the Ephemeral Configuration Database on page 237](#)

PART 4

Requesting Operational and Configuration Information Using the Junos XML Protocol

- [Requesting Operational Information Using the Junos XML Protocol on page 255](#)
- [Requesting Configuration Information Using the Junos XML Protocol on page 269](#)

Requesting Operational Information Using the Junos XML Protocol

- [Requesting Operational Information Using the Junos XML Protocol on page 255](#)
- [Specifying the Output Format for Operational Information Requests in a Junos XML Protocol Session on page 257](#)

Requesting Operational Information Using the Junos XML Protocol

Within a Junos XML protocol session, a client application can request information about the current status of a device running Junos OS. To request operational information, a client application emits the specific request tag element from the Junos XML API that returns the desired information. For example, the `<get-interface-information>` tag corresponds to the `show interfaces` command, the `<get-chassis-inventory>` tag requests the same information as the `show chassis hardware` command, and the `<get-system-inventory>` tag requests the same information as the `show software information` command.

For complete information about the operational request tag elements available in the current Junos OS release, see “Mapping Between Operational Tag Elements, Perl Methods, and CLI Commands” and “Summary of Operational Request Tag Elements” in the *Junos XML API Operational Developer Reference*.

The application encloses the request tag in an `<rpc>` element. The syntax depends on whether the corresponding CLI command has any options included.

```
<rpc>
  <!-- If the command does not have options -->
  <operational-request/>

  <!-- If the command has options -->
  <operational-request>
    <!-- tag elements representing the options -->
  </operational-request>
</rpc>
```

The client application can specify the formatting of the information returned by the Junos XML protocol server. By setting the optional **format** attribute in the opening operational request tag, a client application can specify the format of the response as either XML-tagged format, which is the default, formatted ASCII text, or JavaScript Object

Notation (JSON). For more information about specifying the format, see [“Specifying the Output Format for Operational Information Requests in a Junos XML Protocol Session”](#) on page 257.



NOTE: When displaying operational or configuration data that contains characters outside the 7-bit ASCII character set, Junos OS escapes and encodes these character using the equivalent UTF-8 decimal character reference. For more information see [“Understanding Character Encoding on Devices Running Junos OS”](#) on page 83.

If the client application requests the output in XML-tagged format, the Junos XML protocol server encloses its response in the specific response tag element that corresponds to the request tag element, which is then enclosed in an **<rpc-reply>** tag element.

```
<rpc-reply xmlns:junos="URL">
  <operational-response xmlns="URL-for-DTD">
    <!-- Junos XML tag elements for the requested information -->
  </operational-response>
</rpc-reply>
```

For XML-tagged format, the opening tag for each operational response includes the **xmlns** attribute to define the XML namespace for the enclosed tag elements that do not have a prefix (such as **junos:**) in their names. The namespace indicates which Junos XML document type definition (DTD) defines the set of tag elements in the response. The Junos XML API defines separate DTDs for operational responses from different software modules. For instance, the DTD for interface information is called **junos-interface.dtd** and the DTD for chassis information is called **junos-chassis.dtd**. The division into separate DTDs and XML namespaces means that a tag element with the same name can have distinct functions depending on which DTD it is defined in.

The namespace is a URL of the following form:

```
http://xml.juniper.net/junos/release-code/junos-category
```

release-code is the standard string that represents the Junos OS release that is running on the Junos XML protocol server device.

category specifies the DTD.

The *Junos XML API Operational Developer Reference* includes the text of the Junos XML DTDs for operational responses.

If the client application requests the output in formatted ASCII text, the Junos XML protocol server encloses its response in an **<output>** tag, which is enclosed in an **<rpc-reply>** tag.

```
<rpc-reply xmlns:junos="URL">
  <output>
    operational-response
  </output>
</rpc-reply>
```


Starting in Junos OS Release 14.2, a client application can request operational and configuration information in JSON format. If the client application requests the output in JSON format, the Junos XML protocol server encloses the JSON data in the `<rpc-reply>` tag element.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  operational-response
</rpc-reply>
```



NOTE: Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.

Release History Table

Release	Description
16.1	Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.
14.2	Starting in Junos OS Release 14.2, a client application can request operational and configuration information in JSON format.

Related Documentation

- [Understanding the Request Procedure in a Junos XML Protocol Session on page 54](#)
- [Specifying the Output Format for Operational Information Requests in a Junos XML Protocol Session on page 257](#)
- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)

Specifying the Output Format for Operational Information Requests in a Junos XML Protocol Session

In a Junos XML protocol session, to request information about a routing, switching, or security platform running Junos OS, a client application encloses a Junos XML request tag element in an `<rpc>` tag element. By setting the optional **format** attribute in the opening operational request tag, the client application can specify the formatting of the output returned by the Junos XML protocol server. Information can be returned as XML-tagged format, JavaScript Object Notation (JSON), or formatted ASCII text. The basic syntax is as follows:

```
<rpc>
  <operational-request format="(ascii | json | text | xml)">
    <!-- tag elements for options -->
  </operational-request>
</rpc>
```

XML Format By default, the Junos XML protocol server returns operational information in XML-tagged format. If the value of the **format** attribute is set to "xml", or if the **format** attribute is omitted, the server returns the response in XML. The following example requests information for the ge-0/3/0 interface. The **format** attribute is omitted.

```
<rpc>
  <get-interface-information>
    <brief/>
    <interface-name>ge-0/3/0</interface-name>
  </get-interface-information>
</rpc>
```

The Junos XML protocol server returns the information in XML-tagged format, which is identical to the output displayed in the CLI when you include the **| display xml** option after the operational mode command.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/11.4R1/junos">
<interface-information
  xmlns="http://xml.juniper.net/junos/11.4R1/junos-interface" junos:style="brief">

  <physical-interface>
    <name>ge-0/3/0</name>
    <admin-status junos:format="Enabled">up</admin-status>
    <oper-status>down</oper-status>
    <link-level-type>Ethernet</link-level-type>
    <mtu>1514</mtu>
    <source-filtering>disabled</source-filtering>
    <speed>1000mbps</speed>
    <bpdu-error>none</bpdu-error>
    <l2pt-error>none</l2pt-error>
    <loopback>disabled</loopback>
    <if-flow-control>enabled</if-flow-control>
    <if-auto-negotiation>enabled</if-auto-negotiation>
    <if-remote-fault>online</if-remote-fault>
    <if-device-flags>
      <ifdf-present/>
      <ifdf-running/>
      <ifdf-down/>
    </if-device-flags>
    <if-config-flags>
      <iff-hardware-down/>
      <iff-snmp-traps/>
      <internal-flags>0x4000</internal-flags>
    </if-config-flags>
    <if-media-flags>
      <ifmf-none/>
    </if-media-flags>
  </physical-interface>
</interface-information>
</rpc-reply>
```

ASCII Format To request that the Junos XML protocol server return operational information as formatted ASCII text instead of tagging it with Junos XML tag elements, the client application includes the **format="text"** or **format="ascii"** attribute in the opening request tag. The client application encloses the request in an **<rpc>** tag element.

```
<rpc>
  <get-interface-information format="(text | ascii)">
```

```

    <brief/>
    <interface-name>ge-0/3/0</interface-name>
  </get-interface-information>
</rpc>

```

When the client application includes the **format="text"** or **format="ascii"** attribute in the request tag, the Junos XML protocol server formats the reply as ASCII text and encloses it in an **<output>** tag element. The **format="text"** and **format="ascii"** attributes produce identical output.

```

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/11.4R1/junos">
  <output>
    Physical interface: ge-0/3/0, Enabled, Physical link is Down
      Link-level type: Ethernet, MTU: 1514, Speed: 1000mbps, Loopback: Disabled,
      Source filtering: Disabled, Flow control: Enabled, Auto-negotiation: Enabled,
      Remote fault: Online
      Device flags   : Present Running Down
      Interface flags: Hardware-Down SNMP-Traps Internal: 0x4000
      Link flags     : None
  </output>
</rpc-reply>

```

The following example shows the equivalent operational mode command executed in the CLI:

```

user@host> show interfaces ge-0/3/0 brief
Physical interface: ge-0/3/0, Enabled, Physical link is Down
  Link-level type: Ethernet, MTU: 1514, Speed: 1000mbps, Loopback: Disabled,
  Source filtering: Disabled,
  Flow control: Enabled, Auto-negotiation: Enabled, Remote fault: Online
  Device flags   : Present Running Down
  Interface flags: Hardware-Down SNMP-Traps Internal: 0x4000
  Link flags     : None

```

The formatted ASCII text returned by the Junos XML protocol server is identical to the CLI output except in cases where the output includes disallowed characters such as '<' (less-than sign), '>' (greater-than sign), and '&' (ampersand). The Junos XML protocol server substitutes these characters with the equivalent predefined entity reference of '<', '>', and '&' respectively.

If the Junos XML API does not define a response tag element for the type of output requested by a client application, the Junos XML protocol server returns the reply as formatted ASCII text enclosed in an **<output>** tag element even if XML-tagged output is requested.

For information about the **<output>** tag element, see the *Junos XML API Operational Developer Reference*.



NOTE: The content and formatting of data within an **<output>** tag element are subject to change, so client applications must not depend on them.

JSON Format Starting in Junos OS Release 14.2, a client application can request operational and configuration information in JSON format. To request that the Junos XML protocol server return operational information using JSON format instead of tagging it with Junos XML tag elements, the client application includes the **format="json"** attribute in the opening request tag. The client application encloses the request in an **<rpc>** tag element.

```
<rpc>
  <get-interface-information format="json">
    <brief/>
    <interface-name>cbp0</interface-name>
  </get-interface-information>
</rpc>
]]>]]>
```

When the client application includes the **format="json"** attribute in the request tag, the Junos XML protocol server formats the reply using JSON.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/16.1R1/junos">
{
  "interface-information" : [
    {
      "attributes" : { "xmlns" :
"http://xml.juniper.net/junos/16.1R1/junos-interface",
        "junos:style" : "brief"
      },
      "physical-interface" : [
        {
          "name" : [
            {
              "data" : "cbp0"
            }
          ],
          "admin-status" : [
            {
              "data" : "up",
              "attributes" : { "junos:format" : "Enabled" }
            }
          ],
          "oper-status" : [
            {
              "data" : "up"
            }
          ],
          "if-type" : [
            {
              "data" : "Ethernet"
            }
          ],
          "link-level-type" : [
            {
              "data" : "Ethernet"
            }
          ],
          "mtu" : [
            {
              "data" : "1514"
            }
          ],
          "speed" : [
            {
```

```

        "data" : "Unspecified"
    },
    "clocking" : [
    {
        "data" : "Unspecified"
    }
    ],
    "if-device-flags" : [
    {
        "ifdf-present" : [
        {
            "data" : [null]
        }
        ],
        "ifdf-running" : [
        {
            "data" : [null]
        }
        ]
    }
    ],
    "ifd-specific-config-flags" : [
    {
        "internal-flags" : [
        {
            "data" : "0x0"
        }
        ]
    }
    ],
    "if-config-flags" : [
    {
        "iff-snmp-traps" : [
        {
            "data" : [null]
        }
        ]
    }
    ]
}
]
}
]
}
</rpc-reply>

```



NOTE: Starting in Junos OS Release 17.3R1, OpenConfig supports the operational state emitted by daemons directly in JSON format in addition to XML format. To configure JSON compact format, specify the following CLI command:

set system export-format state-data json compact.

This CLI command converts XML format to compact JSON format. Else, it emits the JSON in non-compact format.

The following example executes the **show system uptime** command and displays the output in non-compact and compact JSON format.

```
user@host> show system uptime | display json
```

Non-compact JSON format:

```
{
  "system-uptime-information" : [
    { "attributes" : { "xmlns" : "http://xml.juniper.net/junos/17.4I0/junos" },
      "current-time" : [
        {
          "date-time" : [
            {
              "data" : "2017-07-14 15:10:24 EDT",
              "attributes" : { "junos:seconds" : "1500059424" }
            }
          ]
        }
      ],
      "time-source" : [
        {
          "data" : " LOCAL CLOCK "
        }
      ],
      "system-booted-time" : [
        {
          "date-time" : [
```

```
{
  "data" : "2017-07-09 21:46:09 EDT",
  "attributes" : {"junos:seconds" : "1499651169"}
},
"time-length" : [
  {
    "data" : "4d 17:24",
    "attributes" : {"junos:seconds" : "408255"}
  }
],
"protocols-started-time" : [
  {
    "date-time" : [
      {
        "data" : "2017-07-09 21:47:21 EDT",
        "attributes" : {"junos:seconds" : "1499651241"}
      }
    ],
    "time-length" : [
      {
        "data" : "4d 17:23",
        "attributes" : {"junos:seconds" : "408183"}
      }
    ]
  }
],
"last-configured-time" : [
  {
```

```
"date-time" : [  
  {  
    "data" : "2017-07-14 15:10:19 EDT",  
    "attributes" : {"junos:seconds" : "1500059419"}  
  },  
],  
"time-length" : [  
  {  
    "data" : "00:00:05",  
    "attributes" : {"junos:seconds" : "5"}  
  },  
],  
"user" : [  
  {  
    "data" : "regress"  
  },  
],  
"uptime-information" : [  
  {  
    "date-time" : [  
      {  
        "data" : "3:10PM",  
        "attributes" : {"junos:seconds" : "1500059424"}  
      },  
    ],  
    "up-time" : [  
      {  
        "data" : "4 days, 17:24",
```



```
"attributes" : {"junos:seconds" : "408285"}
}
],
"active-user-count" : [
{
"data" : "1",
"attributes" : {"junos:format" : "1 user"}
}
],
"load-average-1" : [
{
"data" : "0.11"
}
],
"load-average-5" : [
{
"data" : "0.08"
}
],
"load-average-15" : [
{
"data" : "0.12"
}
]
}
]
```

Compact JSON format for the same command:

```
{
  "system-uptime-information" :
  {
    "current-time" :
    {
      "date-time" : "2017-07-14 15:09:50 EDT"
    },
    "time-source" : " LOCAL CLOCK ",
    "system-booted-time" :
    {
      "date-time" : "2017-07-09 21:46:09 EDT",
      "time-length" : "4d 17:23"
    },
    "protocols-started-time" :
    {
      "date-time" : "2017-07-09 21:47:21 EDT",
      "time-length" : "4d 17:22"
    },
    "last-configured-time" :
    {
      "date-time" : "2017-07-11 23:50:34 EDT",
      "time-length" : "2d 15:19",
      "user" : "regress"
    },
    "uptime-information" :
    {
      "date-time" : "3:09PM",
      "up-time" : "4 days, 17:24",
      "active-user-count" : "1",
      "load-average-1" : "0.00",
      "load-average-5" : "0.05",
```

```
"load-average-15" : "0.11"
}
}
}
```

Release History Table

Release	Description
17.3R1	Starting in Junos OS Release 17.3R1, OpenConfig supports the operational state emitted by daemons directly in JSON format in addition to XML format. To configure JSON compact format, specify the following CLI command: set system export-format state-data json compact. This CLI command converts XML format to compact JSON format. Else, it emits the JSON in non-compact format.
14.2	Starting in Junos OS Release 14.2, a client application can request operational and configuration information in JSON format.

Related Documentation

- [Understanding the Request Procedure in a Junos XML Protocol Session on page 54](#)
- [Requesting Operational Information Using the Junos XML Protocol on page 255](#)
- [Specifying the Output Format for Configuration Data in a Junos XML Protocol Session on page 276](#)

Requesting Configuration Information Using the Junos XML Protocol

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session on page 272](#)
- [Specifying the Output Format for Configuration Data in a Junos XML Protocol Session on page 276](#)
- [Requesting Commit-Script-Style XML Configuration Data Using the Junos XML Protocol on page 281](#)
- [Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol on page 283](#)
- [Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol on page 295](#)
- [Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol on page 298](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using the Junos XML Protocol on page 304](#)
- [Requesting All Configuration Objects of a Specific Type Using the Junos XML Protocol on page 305](#)
- [Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol on page 307](#)
- [Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol on page 310](#)
- [Requesting a Single Configuration Object Using the Junos XML Protocol on page 312](#)
- [Requesting Subsets of Configuration Objects Using Regular Expressions on page 314](#)
- [Requesting Multiple Configuration Elements Using the Junos XML Protocol on page 317](#)
- [Retrieving a Previous \(Rollback\) Configuration Using the Junos XML Protocol on page 319](#)
- [Retrieving the Rescue Configuration Using the Junos XML Protocol on page 321](#)

- [Comparing the Active or Candidate Configuration to a Prior Version Using the Junos XML Protocol on page 324](#)
- [Comparing Two Previous \(Rollback\) Configurations Using the Junos XML Protocol on page 325](#)
- [Requesting an XML Schema for the Configuration Hierarchy Using the Junos XML Protocol on page 327](#)

Requesting Configuration Data Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to request configuration data for a routing, switching, or security platform, a client application encloses the `<get-configuration>` element in an `<rpc>` tag. By setting optional attributes, the client application can specify the source and formatting of the configuration information returned by the Junos XML protocol server. By including the appropriate optional child tag elements, the application can request the entire configuration or specific portions of the configuration. The basic syntax is as follows:

```
<rpc>
  <!-- If requesting the complete configuration -->
  <get-configuration [optional attributes]/>

  <!-- If requesting part of the configuration -->
  <get-configuration [optional attributes]>
    <configuration>
      <!-- tag elements representing the data to return -->
    </configuration>
  </get-configuration>
</rpc>
```



NOTE: To view configuration data in a specific instance of the ephemeral configuration database, a client application must open the ephemeral instance using the `<open-configuration>` operation with the appropriate child tags before executing the `<get-configuration>` request.



NOTE: Starting in Junos OS Release 13.1, within a Junos XML protocol session, a logical system user can use the Junos XML protocol `<get-configuration>` operation to request specific logical system configuration hierarchies using child configuration tags as well as request the entire logical system configuration. When requesting the entire logical system configuration, the RPC reply includes the `<configuration>` root tag. Prior to Junos OS Release 13.1, the `<configuration>` root tag was omitted.

The Junos XML protocol server encloses its reply in an `<rpc-reply>` tag element. It includes attributes with the `junos:` prefix in the opening `<configuration>` tag to indicate when the configuration was last changed or committed and the user who committed it (the attributes appear on multiple lines in the syntax statement only for legibility). For more information about the attributes, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
```

```

<!-- If the application requests Junos XML tag elements -->
<configuration junos:(changed | commit)-seconds="seconds" \
    junos:(changed | commit)-localtime="YYYY-MM-DD hh:mm:ss TZ" \
    [junos:commit-user="username"]>
    <!-- Junos XML tag elements representing configuration elements -->
</configuration>

<!-- If the application requests formatted ASCII text -->
<configuration-text>
    <!-- formatted ASCII configuration statements -->
</configuration-text>

<!-- If the application requests configuration mode set commands -->
<configuration-set>
    <!-- configuration mode commands -->
</configuration-set>

<!-- If the application requests JSON format -->
<configuration-json>
    <!-- JSON configuration data -->
</configuration-json>
</rpc-reply>

```

If a Junos XML tag element is returned within an **<undocumented>** tag element, the corresponding configuration element is not documented in the Junos OS configuration guides or officially supported by Juniper Networks. Most often, the enclosed element is used for debugging only by support personnel. In a smaller number of cases, the element is no longer supported or has been moved to another area of the configuration hierarchy, but appears in the current location for backward compatibility.



NOTE: When displaying operational or configuration data that contains characters outside the 7-bit ASCII character set, Junos OS escapes and encodes these characters using the equivalent UTF-8 decimal character reference. For more information see [“Understanding Character Encoding on Devices Running Junos OS” on page 83](#).

For reference pages for the **<configuration>**, **<configuration-json>**, **<configuration-set>**, **<configuration-text>**, and **<undocumented>** tag elements, see the *Junos XML API Operational Developer Reference*.



NOTE: If the client application locks the candidate configuration before making requests, it needs to unlock it after making its read requests. Other users and applications cannot change the configuration while it remains locked. For more information, see [“Locking and Unlocking the Candidate Configuration or Creating a Private Copy Using the Junos XML Protocol” on page 86](#).

The following topics describe how a client application specifies the source, format, and amount of information returned by the Junos XML protocol server:

- [Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session on page 272](#)
- [Specifying the Output Format for Configuration Data in a Junos XML Protocol Session on page 276](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)

Applications can also request other configuration-related information, including an XML schema representation of the configuration hierarchy or information about previously committed configurations. For more information, see the following:

- [Retrieving a Previous \(Rollback\) Configuration Using the Junos XML Protocol on page 319](#)
- [Retrieving the Rescue Configuration Using the Junos XML Protocol on page 321](#)
- [Comparing the Active or Candidate Configuration to a Prior Version Using the Junos XML Protocol on page 324](#)
- [Comparing Two Previous \(Rollback\) Configurations Using the Junos XML Protocol on page 325](#)
- [Requesting an XML Schema for the Configuration Hierarchy Using the Junos XML Protocol on page 327](#)

**Related
Documentation**

- [Understanding the Request Procedure in a Junos XML Protocol Session on page 54](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Requesting Operational Information Using the Junos XML Protocol on page 255](#)
- [<get-configuration> on page 109](#)

Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session

In a Junos XML protocol session with a device running Junos OS, a client application uses the **<get-configuration>** tag element to request configuration data from the device. A client application can request information from the candidate configuration, the active configuration (that is, the one most recently committed on the device), or from an open instance of the ephemeral configuration database.

The client application can instruct the device to return configuration information from either the candidate configuration or the active configuration by setting the **database** attribute to the appropriate value. To request information from the candidate configuration, a client application includes the **<get-configuration>** tag element and either includes the **database="candidate"** attribute or omits the attribute completely. The Junos XML protocol server returns Junos XML-tagged output by default, except when the **compare** attribute is included.



NOTE: Output of the configuration comparison in XML format using the **compare** filter is available beginning in Junos OS Release 15.1 and is only supported from the CLI.

```
<rpc>
  <get-configuration/>

<!-- OR -->

  <get-configuration>
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

To request information from the active configuration, a client application includes the **<get-configuration>** tag element with the **database="committed"** attribute enclosed within the **<rpc>** element.

```
<rpc>
  <get-configuration database="committed"/>

<!-- OR -->

  <get-configuration database="committed">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

For information about the tag elements to enclose in the **<get-configuration>** tag element, see [“Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session” on page 302](#).



NOTE: Starting in Junos OS Release 13.1, when a logical system user requests the entire logical system configuration using the **<get-configuration>** operation, the RPC reply includes the **<configuration>** root tag. Prior to Junos OS Release 13.1, the **<configuration>** root tag was omitted.

To request information from a specific instance of the ephemeral configuration database, a client application first opens the ephemeral instance using the **<open-configuration>** operation with the appropriate child tags.

```
<rpc>
  <!-- Default instance -->
  <open-configuration>
    <ephemeral/>
```

```
</open-configuration>

<!-- Named instance -->
<open-configuration>
  <ephemeral-instance>instance-name</ephemeral-instance>
</open-configuration>
</rpc>
```

While the ephemeral instance is open, the client application requests information from that instance by using the **<get-configuration>** operation. After all operations on the ephemeral instance are complete, the client application closes the instance with the **<close-configuration/>** operation.

```
<rpc>
  <get-configuration/>

  <!-- OR -->

  <get-configuration>
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
<rpc>
  <close-configuration/>
</rpc>
```

The Junos XML protocol server encloses its reply in the **<rpc-reply>** tag element. Within the **<rpc-reply>** element, the configuration data for the requested configuration is enclosed within the **<configuration>** element for Junos XML-tagged output, the **<configuration-text>** element for formatted ASCII output, the **<configuration-set>** element for configuration mode **set** commands, or the **<configuration-json>** element for configuration data represented using JavaScript Object Notation (JSON).

When returning information from the candidate configuration or from an instance of the ephemeral configuration database, the output includes information about when the configuration was last changed. When displaying the configuration as Junos XML tag elements, the Junos XML protocol server includes this information as attributes in the opening **<configuration>** tag (they appear on multiple lines here only for legibility).

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
    junos:changed-localtime="YYYY-MM-DD hh:mm:ss TZ">
    <!-- Junos XML tag elements representing configuration elements -->
  </configuration>
</rpc-reply>
```

junos:changed-localtime represents the time of the last change as the date and time in the device's local time zone.

junos:changed-seconds represents the time of the last change as the number of seconds since midnight on 1 January 1970.

When returning information from the active configuration, the output includes information about when the configuration was last committed. When displaying the configuration as Junos XML tag elements, the Junos XML protocol server includes this information as attributes in the opening **<configuration>** tag (they appear on multiple lines here only for legibility).

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:commit-seconds="seconds" \
    junos:commit-localtime="YYYY-MM-DD hh:mm:ss TZ" \
    junos:commit-user="username">
    <!-- Junos XML tag elements representing configuration elements -->
  </configuration>
</rpc-reply>
```

junos:commit-localtime represents the commit time as the date and time in the device's local time zone.

junos:commit-seconds represents the commit time as the number of seconds since midnight on 1 January 1970.

junos:commit-user specifies the Junos OS username of the user who requested the commit operation.

The **database** attribute in the application's request can be combined with one or more of the following attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **changed**, which is described in [“Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol”](#) on page 298
- **commit-scripts**, which is described in [“Requesting Commit-Script-Style XML Configuration Data Using the Junos XML Protocol”](#) on page 281
- **compare**, which is described in [“Comparing the Active or Candidate Configuration to a Prior Version Using the Junos XML Protocol”](#) on page 324
- **format**, which is described in [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session”](#) on page 276
- **inherit** and optionally **groups** and **interface-ranges**, which are described in [“Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol”](#) on page 283

The application can also include the **database** attribute after requesting an indicator for identifiers (as described in [“Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol”](#) on page 295).

The following example shows how to request the entire committed configuration. In actual output, the *Junos-version* variable is replaced by a value such as 18.1R1 for the initial version of Junos OS Release 18.1.

Client Application

```
<rpc>
  <get-configuration database="committed"/>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <configuration \
    junos:commit-seconds="seconds" \
    junos:commit-localtime="timestamp" \
    junos:commit-user="username">
    <version>Junos-version</version>
    <system>
      <host-name>big-router</host-name>
      <!-- other children of <system> -->
    </system>
    <!-- other children of <configuration> -->
  </configuration>
</rpc-reply>
```

T1185

Release History Table

Release	Description
13.1	Starting in Junos OS Release 13.1, when a logical system user requests the entire logical system configuration using the <get-configuration> operation, the RPC reply includes the <configuration> root tag.

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Output Format for Configuration Data in a Junos XML Protocol Session on page 276](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Retrieving a Previous \(Rollback\) Configuration Using the Junos XML Protocol on page 319](#)
- [Retrieving the Rescue Configuration Using the Junos XML Protocol on page 321](#)

Specifying the Output Format for Configuration Data in a Junos XML Protocol Session

In a Junos XML protocol session with a device running Junos OS, to request information about a configuration on a routing, switching, or security platform, a client application encloses the **<get-configuration>** element in an **<rpc>** tag. The client application can specify the formatting of the configuration data returned by the Junos XML protocol server by setting optional attributes in the **<get-configuration>** tag.

To request that the Junos XML protocol server return configuration information in Junos XML-tagged output, the client application either includes the **format="xml"** attribute in the `<get-configuration/>` tag or opening `<get-configuration>` tag or omits the attribute completely. The Junos XML protocol server returns Junos XML-tagged output by default, except when the **compare** attribute is included.



NOTE: Output of the configuration comparison in XML format using the **compare** filter is available beginning in Junos OS Release 15.1 and is only supported from the CLI.

```
<rpc>
  <get-configuration/>

  <!-- OR -->

  <get-configuration>
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

To request that the Junos XML protocol server return configuration information as formatted ASCII text instead of tagging it with Junos XML tag elements, the client application includes the **format="text"** attribute in the `<get-configuration/>` tag or opening `<get-configuration>` tag.

```
<rpc>
  <get-configuration format="text"/>

  <!-- OR -->

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

Starting in Junos OS Release 15.1, the client application includes the **format="set"** attribute in the `<get-configuration/>` tag or opening `<get-configuration>` tag to request that the Junos XML protocol server return configuration information as configuration mode **set** commands instead of Junos XML tag elements. The client application encloses the request in an `<rpc>` tag element.

```
<rpc>
  <get-configuration format="set"/>

  <!-- OR -->

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

Starting in Junos OS Release 14.2, a client application can request that the Junos XML protocol server return configuration information in JavaScript Object Notation (JSON) format. To return configuration data in JSON format instead of tagging it with Junos XML tag elements, the client application includes the **format="json"** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag. The client application encloses the request in an **<rpc>** tag element.

```
<rpc>
  <get-configuration format="json"/>

  <!-- OR -->

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

For information about the tag elements to enclose in the **<get-configuration>** tag element, see [“Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session” on page 302](#).



NOTE: Regardless of which output format they request, client applications use Junos XML tag elements to represent the configuration elements to display. The **format** attribute controls the format of the Junos XML protocol server’s output only.

When the application requests Junos XML tag elements, the Junos XML protocol server encloses its output in **<rpc-reply>** and **<configuration>** tag elements. For information about the attributes in the opening **<configuration>** tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- Junos XML tag elements representing configuration elements -->
  </configuration>
</rpc-reply>
```

When the application requests formatted ASCII output, the Junos XML protocol server formats its response in the same way that the CLI **show configuration** command displays configuration data—it uses the newline character, tabs, braces, and square brackets to indicate the hierarchical relationships between configuration statements. The server encloses formatted ASCII configuration statements in **<rpc-reply>** and **<configuration-text>** tag elements.

```
<rpc-reply xmlns:junos="URL">
  <configuration-text>
    <!-- formatted ASCII configuration statements -->
  </configuration-text>
</rpc-reply>
```

When the application requests configuration mode **set** commands, the Junos XML protocol server formats its response in the same way that the CLI **show configuration | display set** command displays configuration data. The server encloses the data in **<rpc-reply>** and **<configuration-set>** tag elements.

```
<rpc-reply xmlns:junos="URL">
```

```
<!-- configuration mode commands -->
</rpc-reply>
```

When the application requests JSON format, the Junos XML protocol server encloses the JSON data in **<rpc-reply>** and **<configuration-json>** tag elements.

```
<rpc-reply xmlns:junos="URL">
  <configuration-json>
    <!-- JSON configuration data -->
  </configuration-json>
</rpc-reply>
```



NOTE: Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.



NOTE: Starting in Junos OS Releases 16.1R4, 16.2R2, and 17.1R1, integers in Junos OS configuration data emitted in JSON format are not enclosed in quotation marks. In earlier releases, integers in JSON configuration data were treated as strings and enclosed in quotation marks.

The **format** attribute can be combined with one or more of the following other attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **commit-scripts** with a value of **commit-scripts="apply"** or **commit-scripts="apply-no-transients"**
- **database**, which is described in [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session”](#) on page 272
- **inherit** and optionally **groups** and **interface-ranges**, which are described in [“Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol”](#) on page 283

It does not make sense to combine the **format="text"** attribute with the **changed** attribute (described in [“Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol”](#) on page 298) or to include it after requesting an indicator for identifiers (described in [“Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol”](#) on page 295). The change and identifier indicators appear only in Junos XML-tagged and JSON output. The **commit-scripts="view"** attribute returns Junos XML-tagged output by default, even if the **format="text"** attribute is included, since this is the format that is input to commit scripts. The **format="xml"** attribute cannot be used with the **compare** attribute, which produces only formatted ASCII output.

An application can request Junos-XML tagged output, formatted ASCII text, configuration mode **set** commands, or JSON output for the entire configuration or any portion of it. For instructions on specifying the amount of data to return, see [“Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session”](#) on page 302.

The following example shows how to request formatted ASCII output from the **[edit policy-options]** hierarchy level in the candidate configuration.

Client Application

```
<rpc>
  <get-configuration format="text">
    <configuration>
      <policy-options/>
    </configuration>
  </get-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <configuration-text>
    policy-options {
      policy-statement load-balancing-policy {
        from {
          route-filter 192.168.10/24 orlonger;
          route-filter 10.114/16 orlonger;
        }
        then {
          load-balance per-packet;
        }
      }
    }
  </configuration-text>
</rpc-reply>
```

T1121

Release History Table

Release	Description
16.1R4	Starting in Junos OS Releases 16.1R4, 16.2R2, and 17.1R1, integers in Junos OS configuration data emitted in JSON format are not enclosed in quotation marks. In earlier releases, integers in JSON configuration data were treated as strings and enclosed in quotation marks.
16.1	Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.
15.1	Starting in Junos OS Release 15.1, the client application includes the format="set" attribute in the <get-configuration/> tag or opening <get-configuration> tag to request that the Junos XML protocol server return configuration information as configuration mode set commands instead of Junos XML tag elements.

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Requesting Commit-Script-Style XML Configuration Data Using the Junos XML Protocol on page 281](#)
- [Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol on page 283](#)
- [Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol on page 295](#)

- [Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol on page 298](#)

Requesting Commit-Script-Style XML Configuration Data Using the Junos XML Protocol

On devices running Junos OS, to view the current configuration in Extensible Markup Language (XML), you issue the **show configuration | display xml** operational mode command. To view the configuration in commit-script-style XML, you use the **show configuration | display commit-scripts view** command. This command displays the configuration in the format that would be input to a commit script.

In a Junos XML protocol session with a device running Junos OS, to request that the server display the configuration as commit-script-style XML data, a client application includes the **commit-scripts="view"** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
  <get-configuration commit-scripts="view"/>

  <!-- OR -->

  <get-configuration commit-scripts="view">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

To view the configuration with commit script changes applied, including both transient and non-transient changes, issue the **show configuration | display commit-scripts** operational mode command on a device running Junos OS. Starting in Junos OS Release 12.1, you can also request this data through the Junos XML protocol server.

To request that the Junos XML protocol server display the configuration with commit script changes applied, including both transient and non-transient changes, a client application includes the **commit-scripts="apply"** attribute in the **<get-configuration/>** tag or opening **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
  <get-configuration commit-scripts="apply"/>

  <!-- OR -->

  <get-configuration commit-scripts="apply">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

To view the configuration with commit script changes applied, but exclude transient changes, issue the **show configuration | display commit-scripts no-transients** operational mode command on a device running Junos OS. Starting in Junos OS Release 12.1, you can also request this data through the Junos XML protocol server.

To request that the Junos XML protocol server display the configuration with commit script changes applied, but exclude transient changes, a client application includes the **commit-scripts="apply-no-transients"** attribute in the `<get-configuration/>` tag or opening `<get-configuration>` tag. It encloses the request in an `<rpc>` tag element:

```
<rpc>
  <get-configuration commit-scripts="apply-no-transients"/>

  <!-- OR -->

  <get-configuration commit-scripts="apply-no-transients">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

The **commit-scripts** attribute can be combined with one or more of the following other attributes in the `<get-configuration/>` tag or opening `<get-configuration>` tag:

- **changed**, which is described in [“Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol”](#) on page 298.
- **database**, which is described in [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session”](#) on page 272.
- **format**, when using **commit-scripts="apply"** or **commit-scripts="apply-no-transients"**.
- **groups**, which is described in [“Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol”](#) on page 283.
- **inherit**, which is described in [“Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol”](#) on page 283.
- **interface-ranges**, which is described in [“Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol”](#) on page 283.
- **junos:key**, which is described in [“Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol”](#) on page 295.

You do not need to include the **changed**, **groups** or **inherit** attributes with the **commit-scripts="view"** attribute. The **commit-scripts-style** XML view includes the **junos:changed="changed"** attribute in the XML tags, and it displays the output with inheritance applied. The tag elements inherited from user-defined groups or interface ranges are displayed within the inheriting tag elements, and the XML tags already include the **junos:group** attribute. To explicitly display the **junos:interface-range** attribute in the **commit-scripts-style** view, you must include the **interface-ranges="interface-ranges"** attribute in the `<get-configuration>` tag.

If you specify a value of **commit-scripts="apply"** or **commit-scripts="apply-no-transients"**, you can specify an output format of formatted ASCII text by also including the **format="text"** attribute.

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol](#) on page 270
- [Specifying the Output Format for Configuration Data in a Junos XML Protocol Session](#) on page 276

- [Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol on page 283](#)
- [Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol on page 295](#)
- [Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol on page 298](#)

Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol

The **<groups>** tag element corresponds to the **[edit groups]** configuration hierarchy. It encloses tag elements representing *configuration groups*, each of which contains a set of configuration statements that are appropriate at multiple locations in the hierarchy. You use the **apply-groups** configuration statement or **<apply-groups>** tag element to insert a configuration group at the appropriate location, achieving the same effect as directly inserting the statements defined in the group. The section of configuration hierarchy to which a configuration group is applied is said to *inherit* the group's statements.

In addition to the groups defined at the **[edit groups]** hierarchy level, Junos OS predefines a group called **junos-defaults**. This group includes configuration statements judged appropriate for basic operations on any routing, switching, or security platform. By default, the statements in this group do not appear in the output of CLI commands that display the configuration, nor in the output returned by the Junos XML protocol server for the **<get-configuration>** tag element. For more information about user-defined configuration groups and the **junos-defaults** group, see the *CLI User Guide*.

The **<interface-range>** tag element corresponds to the **[edit interfaces interface-range]** configuration hierarchy. An interface range is a set of interfaces to which you can apply a common configuration profile. If an interface is a member of an interface range, it inherits the configuration statements set for that range.

The following sections explain how to display groups and interface range configurations within their inheriting elements in configuration data that is requested through a Junos XML protocol session. The sections also discuss how to view the source group or interface range for configuration elements that are inherited from a group or interface range.

- [Specifying Whether Configuration Groups and Interface Ranges Are Inherited or Displayed Separately on page 283](#)
- [Displaying the Source Group for Inherited Configuration Group Elements on page 286](#)
- [Displaying the Source Interface Range for Inherited Configuration Elements on page 288](#)
- [Examples: Specifying Output Format for Configuration Groups on page 291](#)

Specifying Whether Configuration Groups and Interface Ranges Are Inherited or Displayed Separately

By default, the Junos XML protocol server displays the tag element for each user-defined configuration group as a child of the **<groups>** tag element, instead of displaying them as children of the elements to which they are applied. Similarly, the server displays the

tag elements for each user-defined interface range as a child of the `<interface-range>` tag element, instead of displaying them as children of the elements that are members of the interface range. This display mode parallels the default behavior of the CLI configuration mode `show` command, which displays `[edit groups]` and `[edit interfaces interface-range]` as separate hierarchies in the configuration.

To request that the Junos XML protocol server not display the `<groups>`, `<apply-groups>`, or `<interface-range>` elements separately, but instead enclose tag elements inherited from user-defined groups or interface ranges within the inheriting tag elements, a client application includes the `inherit="inherit"` attribute in the `<get-configuration>` tag. It encloses the request in an `<rpc>` tag element:

```
<rpc>
  <get-configuration inherit="inherit"/>

  <!-- OR -->

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

To request that the Junos XML protocol server include tag elements that are inherited from the `junos-defaults` group as well as user-defined configuration groups and interface-ranges, the client application includes the `inherit="defaults"` attribute in the `<get-configuration>` tag.

```
<rpc>
  <get-configuration inherit="defaults"/>

  <!-- OR -->

  <get-configuration inherit="defaults">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

For information about the tag elements to enclose in the `<get-configuration>` tag element, see [“Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session” on page 302](#).

When the client includes the `inherit="inherit"` attribute, the output includes the same information as the output from the following CLI configuration mode command. The output does not include configuration elements inherited from the `junos-defaults` group.

```
user@host# show | display inheritance | except ##
```

When the client includes the `inherit="defaults"` attribute, the output includes the same information as the output from the following CLI configuration mode command:

```
user@host# show | display inheritance defaults | except ##
```

In both cases, the Junos XML protocol server encloses its output in the `<rpc-reply>` tag element and one of the following tag elements depending on the requested format: the `<configuration>` tag element (for Junos XML-tagged output), the `<configuration-text>` tag element (for formatted ASCII output), the `<configuration-set>` tag element (for configuration mode set commands), or the `<configuration-json>` tag element (for JSON-formatted data). For information about the attributes in the opening `<configuration>` tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- Junos XML tag elements representing configuration elements -->
  </configuration>

  <!-- OR -->

  <configuration-text>
    <!-- formatted ASCII configuration statements -->
  </configuration-text>

  <!-- OR -->

  <configuration-set>
    <!-- configuration mode commands -->
  </configuration-set>

  <!-- OR -->

  <configuration-json>
    <!-- JSON-formatted configuration data -->
  </configuration-json>
</rpc-reply>
```

The `inherit` attribute can be combined with one or more of the following attributes in the `<get-configuration/>` tag or opening `<get-configuration>` tag:

- **changed**, which is described in [“Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol” on page 298](#)
- **database**, which is described in [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#)
- **format**, which is described in [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#)
- **groups**, which is described in [“Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol” on page 283](#)
- **interface-ranges**, which is described in [“Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol” on page 283](#)
- **junos:key**, which is described in [“Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol” on page 295](#).

The application can also include the `inherit` attribute after requesting an indicator for identifiers (as described in [“Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol” on page 295](#)).

Displaying the Source Group for Inherited Configuration Group Elements

To request that the Junos XML protocol server indicate the configuration group from which each configuration element is inherited, a client application combines the **inherit** attribute with the **groups="groups"** attribute in the **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
  <get-configuration inherit="(defaults | inherit)" groups="groups"/>

  <!-- OR -->

  <get-configuration inherit="(defaults | inherit)" groups="groups">
    <!-- tag elements indicating the configuration elements to return
  -->
  </get-configuration>
</rpc>
```

For information about the tag elements to enclose in the **<get-configuration>** tag element, see ["Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session" on page 302](#).

When you include both the **inherit** and **groups="groups"** attributes in the request, the Junos XML protocol server displays each configuration group element within its inheriting element, and the inherited element then includes information that indicates the source group. The format for the output determines how the source group information is displayed in the resulting configuration.

If the output is tagged with Junos XML tag elements (the **format="xml"** attribute is included or the **format** attribute is omitted), the Junos XML protocol server includes the **junos:group="source-group"** attribute in the opening tags of configuration elements that are inherited from configuration groups. The response is enclosed in **<configuration>** and **<rpc-reply>** tag elements.

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- For each inherited element -->
    <!-- opening-tags-for-parents-of-the-element -->
    <inherited-element junos:group="source-group">
      <inherited-child-of-inherited-element
junos:group="source-group">
        <!-- inherited-children-of-child junos:group="source-group"
-->
      </inherited-child-of-inherited-element>
    </inherited-element>
    <!-- closing-tags-for-parents-of-the-element -->
  </configuration>
</rpc-reply>
```

If the output is formatted ASCII text (the **format="text"** attribute is included), the Junos XML protocol server inserts three commented lines with the information about the source group immediately above each inherited element. The response is enclosed in **<configuration-text>** and **<rpc-reply>** tag elements.

```
<rpc-reply xmlns:junos="URL">
  <configuration-text>
  /* For each inherited element */
    /* parent levels for the element */
```

```

##
## 'inherited-element' was inherited from group 'source-group'
##
inherited-element {
  ##
  ## 'inherited-child' was inherited from group 'source-group'
  ##
  inherited-child {
    ... child statements of inherited-child ...
  }
}
/* closing braces for parent levels of the element */
</configuration-text>
</rpc-reply>

```

If the output is in JSON format (the **format="json"** attribute is included), the Junos XML protocol server includes the **"junos:group": "source-group"** attribute in the attribute list for the inherited element. The response is enclosed in **<configuration-json>** and **<rpc-reply>** tag elements.

```

<rpc-reply xmlns:junos="URL">
  <configuration-json>
  {
    "configuration" : {
      /* JSON objects for parent levels of the element */
      "inherited-child" : {
        "@" : {
          "junos:group" : "source-group"
        },
        "inherited-object" : [
          {
            "@" : {
              "junos:group" : "source-group"
            },
            "name" : "identifier"
          }
        ],
        "@inherited-statement" : {
          "junos:group" : "source-group"
        }
      }
    }
  }
  /* closing braces for parent levels of the element */
}
</configuration-json>
</rpc-reply>

```



NOTE: Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization. The new default uses the "@" symbol instead of the field name "attribute" to indicate an attribute.

When the **groups="groups"** attribute is combined with the **inherit="inherit"** attribute, the XML output includes the same information as the output from the following CLI configuration mode command. The output does not include configuration elements inherited from the **junos-defaults** group:

```
user@host# show | display inheritance | display xml groups
```

When the **groups="groups"** attribute is combined with the **inherit="defaults"** attribute, the XML output includes the same information as the output from the following CLI configuration mode command:

```
user@host# show | display inheritance defaults | display xml groups
```

The **inherit** and **groups** attributes can be combined with one or more of the following other attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **changed**, which is described in “Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol” on page 298.
- **database**, which is described in “Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272.
- **format**, which is described in “Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276. The application can request either Junos XML-tagged, formatted ASCII, or JSON output.
- **interface-ranges**, which is described in “Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol” on page 283.
- **junos:key**, which is described in “Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol” on page 295.

The application can also include the **inherit** and **groups** attributes after requesting an indicator for identifiers (as described in “Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol” on page 295).

Displaying the Source Interface Range for Inherited Configuration Elements

To request that the Junos XML protocol server indicate the interface range from which each configuration element is inherited, a client application combines the **inherit** attribute with the **interface-ranges="interface-ranges"** attribute in the **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
  <get-configuration inherit="inherit" interface-ranges="interface-ranges"/>

  <!-- OR -->

  <get-configuration inherit="inherit" interface-ranges="interface-ranges">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

For information about the tag elements to enclose in the **<get-configuration>** tag element, see “Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session” on page 302.

When you include both the **inherit** and **interface-ranges="interface-ranges"** attributes in the request, the Junos XML protocol server displays each interface range configuration element within its inheriting element, and the inherited element then includes information that indicates the source interface range. The format for the output determines how the source interface range information is displayed in the resulting configuration.

If the output is tagged with Junos XML tag elements (the **format="xml"** attribute is included or the **format** attribute is omitted), the Junos XML protocol server includes the **junos:interface-range="source-interface-range"** attribute in the opening tags of configuration elements that are inherited from an interface range. The response is enclosed in **<configuration>** and **<rpc-reply>** tag elements.

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <interfaces>
      <!-- For each inherited element -->
      <interface junos:interface-range="source-interface-range">
        <inherited-element
junos:interface-range="source-interface-range">
          <inherited-child-of-inherited-element
            junos:interface-range="source-interface-range">
              <!-- inherited-children-of-child
                junos:interface-range="source-interface-range"
              -->
            </inherited-child-of-inherited-element>
          </inherited-element>
        </interface>
      </interfaces>
    </configuration>
  </rpc-reply>
```

If the output is formatted ASCII text (the **format="text"** attribute is included), the Junos XML protocol server inserts three commented lines with the information about the source interface range immediately above each inherited element. The response is enclosed in **<configuration-text>** and **<rpc-reply>** tag elements.

```
<rpc-reply xmlns:junos="URL">
  <configuration-text>
    interfaces {
      <!-- For each inherited element -->
      ##
      ## 'interface-name' was expanded from interface-range 'source-interface-range'
      ##
      interface-name {
        ##
        ## 'inherited-element' was expanded from interface-range 'source-interface-range'

        ##
        inherited-element {
          inherited-child {
            ... child statements of inherited-child ...
          }
        }
      }
    }
  </configuration-text>
</rpc-reply>
```

If the output is in JSON format (the **format="json"** attribute is included), the Junos XML protocol server includes the **"junos:interface-range" : "source-interface-range"** attribute in the attribute list for the inherited element. The response is enclosed in **<configuration-json>** and **<rpc-reply>** tag elements.

```
<rpc-reply xmlns:junos="URL">
  <configuration-json>
  {
    "configuration" : {
      /* JSON objects for parent levels of the element */
      "inherited-child" : {
        "@" : {
          "junos:interface-range" : "source-interface-range"
        },
        "inherited-object" : [
          {
            "@" : {
              "junos:interface-range" : "source-interface-range"
            },
            "name" : "identifier"
          }
        ],
        "@inherited-statement" : {
          "junos:interface-range" : "source-interface-range"
        }
      }
      /* closing braces for parent levels of the element */
    }
  }
</configuration-json>
</rpc-reply>
```



NOTE: Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization. The new default uses the "@" symbol instead of the field name "attribute" to indicate an attribute.

When the **interface-ranges="interface-ranges"** attribute is combined with the **inherit="inherit"** attribute, the XML output includes the same information as the output from the following CLI configuration mode command:

```
user@host# show | display inheritance | display xml interface-ranges
```

The **inherit** and **interface-ranges** attributes can be combined with one or more of the following other attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag:

- **changed**, which is described in [“Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol”](#) on page 298.
- **database**, which is described in [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session”](#) on page 272.
- **format**, which is described in [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session”](#) on page 276. The application can request Junos XML-tagged, formatted ASCII, or JSON output.

- **groups**, which is described in “Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol” on page 283.
- **junos:key**, which is described in “Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol” on page 295.

The application can also include the **inherit** and **interface-ranges** attributes after requesting an indicator for identifiers (as described in “Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol” on page 295).

Examples: Specifying Output Format for Configuration Groups

The following sample configuration hierarchy defines a configuration group called **interface-group**. The **apply-groups** statement applies the statements in the group at the **[edit interfaces]** hierarchy level:

```
[edit]
groups {
  interface-group {
    interfaces {
      so-1/1/1 {
        encapsulation ppp;
      }
    }
  }
}
apply-groups interface-group;
interfaces {
  fxp0 {
    unit 0 {
      family inet {
        address 192.168.4.207/24;
      }
    }
  }
}
```

When the **inherit** attribute is not included in the **<get-configuration/>** tag, the output includes the **<groups>** and **<apply-groups>** tag elements as separate items. The **<groups>** tag element encloses the tag elements defined in the **interface-group** configuration group. The placement of the **<apply-groups>** tag element directly above the **<interfaces>** tag element indicates that the **[edit interfaces]** hierarchy inherits the statements defined in the **interface-group** configuration group.

Client Application Junos XML Protocol Server

```

<rpc>
  <get-configuration/>
</rpc>

<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds='seconds' \
    junos:changed-localtime='timestamp'>
    <groups>
      <name>interface-group</name>
      <interfaces>
        <interface>
          <name>so-1/1/1</name>
          <encapsulation>ppp</encapsulation>
        </interface>
      </interfaces>
    </groups>
    <apply-groups>interface-group</apply-groups>
    <interfaces>
      <interface>
        <name>fxp0</name>
        <unit>
          <name>0</name>
          <family>
            <inet>
              <address>
                <name>192.168.4.207/24</name>
              </address>
            </inet>
          </family>
        </unit>
      </interface>
    </interfaces>
  </configuration>
</rpc-reply>

```

T-1188

When the **inherit** attribute is included in the **<get-configuration/>** tag, the **<interfaces>** tag element encloses the tag elements defined in the **interface-group** configuration group. The **<groups>** and **<apply-groups>** tag elements are not displayed.

Client Application Junos XML Protocol Server

```

<rpc>
  <get-configuration inherit="inherit"/>
</rpc>

<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds='seconds' \
    junos:changed-localtime='timestamp'>
    <interfaces>
      <interface>
        <name>fxp0</name>
        <unit>
          <name>0</name>
          <family>
            <inet>
              <address>
                <name>192.168.4.207/24</name>
              </address>
            </inet>
          </family>
        </unit>
      </interface>
      <interface>
        <name>so-1/1/1</name>
        <encapsulation>ppp</encapsulation>
      </interface>
    </interfaces>
  </configuration>
</rpc-reply>

```

T1189

When the **groups="groups"** attribute is combined with the **inherit** attribute in the **<get-configuration/>** tag, the **<interfaces>** tag element encloses the tag elements defined in the **interface-group** configuration group, which are marked with the **junos:group="interface-group"** attribute.

Client Application Junos XML Protocol Server

```
<rpc>
  <get-configuration inherit="inherit" groups="groups"/>
</rpc>

<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
    junos:changed-localtime="timestamp">
    <interfaces>
      <interface>
        <name>fxp0</name>
        <unit>
          <name>0</name>
          <family>
            <inet>
              <address>
                <name>192.168.4.207/24</name>
              </address>
            </inet>
          </family>
        </unit>
      </interface>
      <interface junos:group="interface-group">
        <name junos:group="interface-group">so-1/1/1</name>
        <encapsulation junos:group="interface-group">ppp</encapsulation>
      </interface>
    </interfaces>
  </configuration>
</rpc-reply>
```

T1190

Release History Table

Release	Description
16.1	Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Output Format for Configuration Data in a Junos XML Protocol Session on page 276](#)
- [Requesting Commit-Script-Style XML Configuration Data Using the Junos XML Protocol on page 281](#)
- [Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol on page 295](#)
- [Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol on page 298](#)

Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to request that the server indicate whether a child configuration element is an identifier for its parent element, a client application includes the `junos:key="key"` attribute in the opening `<junoscript>` tag for the Junos XML protocol session or includes the `junos:key="key"` or `key="key"` attribute in the `<get-configuration>` request tag:

```
<junoscript version="version" hostname="hostname" junos:key="key"
  release="release-code">

  <!-- OR -->

  <rpc>
    <get-configuration (junos:key | key)="key">
      <!-- tag elements for the configuration elements to return -->
    </get-configuration>
  </rpc>
```

For more information about the `<junoscript>` tag, see [“Starting Junos XML Protocol Sessions” on page 70](#).

When the identifier indicator is requested, the Junos XML protocol server includes the `junos:key="key"` attribute in the opening tag for each identifier. As always, the Junos XML protocol server encloses its response in `<rpc-reply>` and `<configuration>` tag elements. In the following example, the identifier tag element is called `<name>`:

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tag for each parent of the object -->

    <!-- For each configuration object with an identifier -->
    <object>
      <name junos:key="key">identifier</name>
      <!-- additional children of object -->
    </object>
    <!-- closing tag for each parent of the object -->

  </configuration>
</rpc-reply>
```

If the requested output format is JSON, the Junos XML protocol server adds a metadata object that includes `"junos:key" : "key"` to indicate the identifier. If the Junos OS object uses `name` for the identifier, a metadata object with name `"@"` is added as a new member of the object. If the Junos OS object uses an identifier other than `name`, the metadata object is added as a sibling name/value pair that uses the name `"@"` concatenated with the identifier name. The response is enclosed in `<configuration-json>` and `<rpc-reply>` tag elements.

```
<rpc-reply xmlns:junos="URL">
  <configuration-json>
  {
    "configuration" : {
      /* JSON objects for parent levels of the element */
      "object" : [
        {
          "@" : {
```

```

        "junos:key" : "key"
      },
      "name" : "identifier",
      "identifier-name" : "identifier-value",
      "@identifier-name" : {
        "junos:key" : "key"
      },
      /* additional data and child objects */ # if any
    }
  ]
  /* closing braces for parent levels of the element */
}
}
</configuration-json>
</rpc-reply>

```

In the following output, the combination of **name** and **next-hop** uniquely identify the static route:

```

{
  "configuration" : {
    "routing-options" : {
      "static" : {
        "route" : [
          {
            "@" : {
              "junos:key" : "key"
            },
            "name" : "172.16.0.0/12",
            "next-hop" : ["198.51.100.1"],
            "@next-hop" : {
              "junos:key" : "key"
            },
            "retain" : [null],
            "no-readvertise" : [null]
          }
        ]
      }
    }
  }
}

```



NOTE: Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization. The new default uses the "@" symbol instead of the field name "attribute" to indicate an attribute.

The client application can include one or more of the following other attributes in the `<get-configuration/>` tag or opening `<get-configuration>` tag when the `junos:key` attribute is included in the opening `<junoscript>` or `<get-configuration>` tags:

- **changed**, which is described in [“Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol”](#) on page 298
- **commit-scripts**, which is described in [“Requesting Commit-Script-Style XML Configuration Data Using the Junos XML Protocol”](#) on page 281

- **database**, which is described in “[Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session](#)” on page 272
- **inherit** and optionally **groups** and **interface-ranges**, which are described in “[Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol](#)” on page 283

When requesting an indicator for identifiers, it does not make sense to include the **format="text"** attribute in the **<get-configuration>** tag element (as described in “[Specifying the Output Format for Configuration Data in a Junos XML Protocol Session](#)” on page 276). The **junos:key="key"** attribute appears only in Junos XML-tagged output, which is the default output format, and in JSON output. The **compare** attribute produces only text output, so when the **compare** attribute is included in the **<get-configuration>** tag, the **junos:key="key"** attribute does not appear in the output.

The following example shows how indicators for identifiers appear on configuration elements at the **[edit interfaces]** hierarchy level in the candidate configuration when the **junos:key="key"** attribute is included in the opening **<junoscript>** tag emitted by the client application for the session. The two opening **<junoscript>** tags appear on multiple lines for legibility only. Neither client applications nor the Junos XML protocol server insert newline characters within tags. Also, for brevity the output includes just one interface, the loopback interface lo0.

Client Application

```
<?xml version="1.0" encoding="us-ascii"?>
<junoscript version="1.0" \
  junos:key="key" \
  release="JUNOS-release">

<rpc>
  <get-configuration>
    <configuration>
      <interfaces/>
    </configuration>
  </get-configuration>
</rpc>
```

Junos XML Protocol Server

```
<?xml version="1.0" encoding="us-ascii"?>
<junoscript version="1.0" hostname="router1" \
  os="JUNOS" release="JUNOS-release" \
  xmlns="URL"xmlns:junos="URL" \
  xmlns:xnm="URL">

<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
    junos:changed-localtime="timestamp">
    <interfaces>
      <!-- tag elements for other interfaces -->
      <interface>
        <name junos:key="key">lo0</name>
        <unit>
          <name junos:key="key">0</name>
          <family>
            <inet>
              <address>
                <name junos:key="key">127.0.0.1/32</name>
              </address>
            </inet>
          </family>
        </unit>
      </interface>
      <!-- tag elements for other interfaces -->
    </interfaces>
  </configuration>
</rpc-reply>
```

T1187

Release History Table

Release	Description
16.1	Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Output Format for Configuration Data in a Junos XML Protocol Session on page 276](#)
- [Requesting Commit-Script-Style XML Configuration Data Using the Junos XML Protocol on page 281](#)
- [Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol on page 283](#)
- [Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol on page 298](#)

Requesting Change Indicators for Configuration Elements Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to request that the server indicate which configuration elements have changed since the last commit, a client application includes the **changed="changed"** attribute in the **<get-configuration/>** request tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
  <get-configuration changed="changed"/>

  <!-- OR -->

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

For information about the tag elements to enclose in the **<get-configuration>** tag element, see ["Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session" on page 302](#).

The configuration source determines which elements are marked as changed. When the **database="candidate"** attribute is included in the **<get-configuration/>** tag or when the **database** attribute is omitted, the candidate configuration is compared to the active configuration. Elements added to the candidate configuration after the last commit operation are marked with the **junos:changed="changed"** attribute. When the **database="committed"** attribute is included in the **<get-configuration/>** tag, the active configuration is compared to the first rollback configuration. Elements added to the active configuration by the most recent commit are marked with the **junos:changed="changed"** attribute.

The Junos XML protocol server indicates which elements have changed by including the `junos:changed="changed"` attribute in the opening tag of every parent tag element in the path to the changed configuration element. If the changed configuration element is represented by a single (empty) tag, the `junos:changed="changed"` attribute appears in the tag. If the changed element is represented by a container tag element, the `junos:changed="changed"` attribute appears in the opening container tag and also in the opening tag for each child tag element enclosed in the container tag element.

The Junos XML protocol server encloses its response in `<rpc-reply>` and `<configuration>` tag elements. For information about the standard attributes in the opening `<configuration>` tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <configuration standard-attributes junos:changed="changed">
    <!-- opening-tag-for-each-parent-level junos:changed="changed" -->

    <!-- For each changed element, EITHER -->
    <element junos:changed="changed"/>

    <!-- OR -->

    <element junos:changed="changed">
      <first-child-of-element junos:changed="changed">
        <second-child-of-element junos:changed="changed">
          <!-- additional children of element -->
        </element>

      <!-- closing-tag-for-each-parent-level -->
    </configuration>
  </rpc-reply>
```

If the requested output format is JSON, the Junos XML protocol server includes the `"junos:changed": "changed"` attribute in the attribute lists for the same elements as described previously and encloses the response in `<configuration-json>` and `<rpc-reply>` tag elements.



NOTE: Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization. The new default uses the "@" symbol instead of the field name "attribute" to indicate an attribute.



NOTE: When a commit operation succeeds, the Junos XML protocol server removes the `junos:changed="changed"` attribute from all tag elements. However, if warnings are generated during the commit, the attribute is not removed. In this case, the `junos:changed="changed"` attribute appears on tag elements that changed before the commit as well as those that changed after the commit.

An example of a commit-time warning is the message explaining that a configuration element will not actually apply until the device is rebooted. The warning appears in the tag string that the Junos XML protocol server returns to confirm the success of the commit, enclosed in an `<xnm:warning>` tag element.

To remove the `junos:changed="changed"` attribute from elements that changed before the commit, the client application must take any action necessary to eliminate the cause of the warning, and commit the configuration again.

The **changed** attribute can be combined with one or more of the following other attributes in the `<get-configuration/>` tag or opening `<get-configuration>` tag:

- **database**, which is described in “[Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session](#)” on page 272. Request change indicators in either the candidate or active configuration.
- **inherit** and optionally **groups** and **interface-ranges**, which are described in “[Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol](#)” on page 283.
- **junos:key**, which is described in “[Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol](#)” on page 295.

It does not make sense to combine the **changed** attribute with the **format="text"** attribute or with the **compare** attribute, which produces only text output. The `junos:changed="changed"` attribute appears only in Junos XML-tagged output, which is the default output format, and in JSON output. When the **commit-scripts="view"** attribute is included in the `<get-configuration>` tag, the `junos:changed="changed"` attribute is automatically included in the output, and you do not need to explicitly include this attribute in the `<get-configuration>` request.

The application can also include the **changed** attribute after requesting an indicator for identifiers (as described in “[Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol](#)” on page 295).

The following example shows how to request change indicators for configuration elements at the `[edit system syslog]` hierarchy level in the candidate configuration. The output indicates that a log file called **interactive-commands** has been configured since the last commit.

Client Application Junos XML Protocol Server

```

<rpc>
  <get-configuration changed="changed">
    <configuration>
      <system>
        <syslog/>
      </system>
    </configuration>
  </get-configuration>
</rpc>

<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
    junos:changed-localtime="timestamp" junos:changed="changed">
    <system junos:changed="changed">
      <syslog junos:changed="changed">
        <file>
          <name>messages</name>
          <contents>
            <name>any</name>
            <info/>
          </contents>
        </file>
        <file junos:changed="changed">
          <name junos:changed="changed">interactive-commands</name>
          <contents>
            <name junos:changed="changed">interactive-commands</name>
            <notice junos:changed="changed"/>
          </contents>
        </file>
      </syslog>
    </system>
  </configuration>
</rpc-reply>

```

T1186

Release History Table

Release	Description
16.1	Starting in Junos OS Release 16.1, devices running Junos OS emit JSON-formatted configuration data using a new default implementation for serialization.

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Output Format for Configuration Data in a Junos XML Protocol Session on page 276](#)
- [Requesting Commit-Script-Style XML Configuration Data Using the Junos XML Protocol on page 281](#)
- [Specifying the Output Format for Configuration Groups and Interface Ranges Using the Junos XML Protocol on page 283](#)
- [Requesting Identifier Indicators for Configuration Elements Using the Junos XML Protocol on page 295](#)

Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session

In a Junos XML protocol session with a device running Junos OS, a client application can request the entire configuration or specific portions of the configuration by emitting the `<get-configuration>` tag element and including the appropriate child tag elements.

For information about requesting different amounts of configuration information, see the following topics:

- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using the Junos XML Protocol on page 304](#)
- [Requesting All Configuration Objects of a Specific Type Using the Junos XML Protocol on page 305](#)
- [Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol on page 307](#)
- [Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol on page 310](#)
- [Requesting a Single Configuration Object Using the Junos XML Protocol on page 312](#)
- [Requesting Subsets of Configuration Objects Using Regular Expressions on page 314](#)
- [Requesting Multiple Configuration Elements Using the Junos XML Protocol on page 317](#)

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session on page 272](#)
- [Retrieving a Previous \(Rollback\) Configuration Using the Junos XML Protocol on page 319](#)
- [Retrieving the Rescue Configuration Using the Junos XML Protocol on page 321](#)
- [Comparing the Active or Candidate Configuration to a Prior Version Using the Junos XML Protocol on page 324](#)
- [Comparing Two Previous \(Rollback\) Configurations Using the Junos XML Protocol on page 325](#)

Requesting the Complete Configuration Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to request the entire candidate configuration or the complete configuration in an open instance of the ephemeral configuration database, a client application encloses the `<get-configuration/>` tag in an `<rpc>` tag element:

```
<rpc>
  <get-configuration/>
</rpc>
```



NOTE: If a client application issues the Junos XML protocol `<open-configuration>` operation to open a specific configuration database before executing the `<get-configuration>` operation, the server returns the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration, unless the active configuration is explicitly requested by including the `database="committed"` attribute.

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested configuration in `<configuration>` and `<rpc-reply>` tag elements. For information about the attributes in the opening `<configuration>` tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- Junos XML tag elements for all configuration elements -->
  </configuration>
</rpc-reply>
```

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII or JSON or an indicator for identifiers), the application can include attributes in the `<get-configuration/>` tag, its opening `<junoscript>` tag, or both. For more information, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#) and [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#).

The following example shows how to request the complete candidate configuration tagged with Junos XML tag elements (the default). In actual output, the `JUNOS-version` variable is replaced by a value such as `18.1R1` for the initial version of Junos OS Release 18.1.

Client Application Junos XML Protocol Server

```
<rpc>
  <get-configuration/>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds='seconds' \
    junos:changed-localtime='timestamp">
    <version>JUNOS-version</version>
    <system>
      <host-name>big-router</host-name>
      <!-- other children of <system>-->
    </system>
    <!-- other children of <configuration>-->
  </configuration>
</rpc-reply>
```

T1191

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)

- [Retrieving a Previous \(Rollback\) Configuration Using the Junos XML Protocol on page 319](#)
- [Retrieving the Rescue Configuration Using the Junos XML Protocol on page 321](#)
- [Comparing the Active or Candidate Configuration to a Prior Version Using the Junos XML Protocol on page 324](#)
- [Comparing Two Previous \(Rollback\) Configurations Using the Junos XML Protocol on page 325](#)

Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to request complete information about all child configuration elements at a hierarchy level or in a container object that does not have an identifier, a client application emits a `<get-configuration>` tag element that encloses the tag elements representing all levels in the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to the the immediate parent level of the level or container object, which is represented by an empty tag. The entire request is enclosed in an `<rpc>` tag element.

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the level -->
        <requested-level/>
      <!-- closing tags for each parent of the level -->
    </configuration>
  </get-configuration>
</rpc>
```

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested section of the configuration in `<configuration>` and `<rpc-reply>` tag elements. For information about the attributes in the opening `<configuration>` tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the level -->
      <hierarchy-level>
        <!-- child tag elements of the level -->
      </hierarchy-level>
    <!-- closing tags for each parent of the level -->
  </configuration>
</rpc-reply>
```

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII or JSON or an indicator for identifiers), the application can include attributes in the opening `<get-configuration>` tag, its opening `<junoscript>` tag, or both. For more information, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#) and [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#).

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same `<get-configuration>` tag element. For more information, see [“Requesting Multiple Configuration Elements Using the Junos XML Protocol” on page 317](#).

The following example shows how to request the contents of the `[edit system login]` hierarchy level in the candidate configuration. The output is tagged with Junos XML tag elements, which is the default.

Client Application

```
<rpc>
  <get-configuration>
    <configuration>
      <system>
        <login/>
      </system>
    </configuration>
  </get-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds="seconds" \
    junos:changed-localtime="timestamp">
    <system>
      <login>
        <user>
          <name>barbara</name>
          <full-name>Barbara Anderson</full-name>
          <!-- other child tags for this user - -->
        </user>
        <!-- other children of <login> - -->
      </login>
    </system>
  </configuration>
</rpc-reply>
```

T1192

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Requesting All Configuration Objects of a Specific Type Using the Junos XML Protocol on page 305](#)
- [Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol on page 307](#)
- [Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol on page 310](#)

Requesting All Configuration Objects of a Specific Type Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to request complete information about all configuration objects of a specified type in a hierarchy level, a client

application emits a **<get-configuration>** tag element that encloses the tag elements representing all levels of the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to the immediate parent level for the object type. An empty tag represents the requested object type. The entire request is enclosed in an **<rpc>** tag element.

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object type -->
      <object-type/>
      <!-- closing tags for each parent of the object type -->
    </configuration>
  </get-configuration>
</rpc>
```

This type of request is useful when the object's parent hierarchy level has child objects of multiple types and the application is requesting just one of the types. If the requested object is the only possible child type, then this type of request yields the same output as a request for the complete parent hierarchy (described in [“Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using the Junos XML Protocol” on page 304](#)).

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested objects in **<configuration>** and **<rpc-reply>** tag elements. For information about the attributes in the opening **<configuration>** tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the object type -->
    <first-object>
      <!-- child tag elements for the first object -->
    </first-object>
    <second-object>
      <!-- child tag elements for the second object -->
    </second-object>
    <!-- additional instances of the object -->
    <!-- closing tags for each parent of the object type -->
  </configuration>
</rpc-reply>
```

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII or JSON or an indicator for identifiers), the application can include attributes in the opening **<get-configuration>** tag, its opening **<junoscript>** tag, or both. For more information, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#) and [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#).

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-configuration>** tag element. For more information, see [“Requesting Multiple Configuration Elements Using the Junos XML Protocol” on page 317](#).

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using the Junos XML Protocol on page 304](#)
- [Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol on page 307](#)
- [Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol on page 310](#)

Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to request information about a specific number of configuration objects of a specific type, a client application emits the **<get-configuration>** tag element and encloses the tag elements that represent all levels of the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to the immediate parent level for the object type. An empty tag represents the requested object type, and the tag includes the following attributes:

- **count** specifies the number of objects to return
- **start** specifies the index number of the first object to return (1 for the first object, 2 for the second, and so on)

If the application is requesting only the first object in the hierarchy, it includes the **count="1"** attribute and omits the **start** attribute. The application encloses the entire request in an **<rpc>** tag element.

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object -->
      <object-type count="count" start="index"/>
      <!-- closing tags for each parent of the object -->
    </configuration>
  </get-configuration>
</rpc>
```



NOTE: The **count** and **start** attributes are not supported when requesting configuration data in JSON format.

The Junos XML protocol server returns the requested objects starting with the object specified by the **start** attribute and running consecutively. When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested objects in **<configuration>** and **<rpc-reply>** tag elements, starting with the object specified by the **start** attribute and running consecutively.

For each object, the server includes two attributes:

- **junos:position**, to specify the object's numerical index
- **junos:total**, to report the total number of such objects that exist in the hierarchy

In the following example Junos XML output, the identifier tag element is called **<name>**. For information about the attributes in the opening **<configuration>** tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the object type -->
    <first-object junos:position="index1" junos:total="total">
      <name>identifier-for-first-object</name>
      <!-- other child tag elements of the first object -->
    </first-object>
    <second-object junos:position="index2" junos:total="total">
      <name>identifier-for-second-object</name>
      <!-- other child tag elements of the second object -->
    </second-object>
    <!-- additional objects -->
    <!-- closing tags for each parent of the object type -->
  </configuration>
</rpc-reply>
```

The **junos:position** and **junos:total** attributes do not appear if the client requests formatted ASCII output by including the **format="text"** attribute in the **<get-configuration>** tag element (as described in [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#)).

To specify the source of the output (candidate or active configuration), the application can include attributes in the opening **<get-configuration>** tag, its opening **<junoscript>** tag, or both. For more information, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-configuration>** tag element. For more information, see [“Requesting Multiple Configuration Elements Using the Junos XML Protocol” on page 317](#).

The following example shows how to request the third and fourth Junos user accounts at the **[edit system login]** hierarchy level. The output is from the candidate configuration and is tagged with Junos XML tag elements (the default).

Client Application

```

<rpc>
  <get-configuration>
    <configuration>
      <system>
        <login>
          <user count="2" start="3"/>
        </login>
      </system>
    </configuration>
  </get-configuration>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds='seconds' \
    junos:changed-localtime='timestamp">
    <system>
      <login>
        <user junos:position="3" junos:total="22">
          <name>barbara</name>
          <uid>1423</uid>
          <class>operator</class>
        </user>
        <user junos:position="4" junos:total="22">
          <name>carlo</name>
          <uid>1426</uid>
          <class>operator</class>
        </user>
      </login>
    </system>
  </configuration>
</rpc-reply>

```

T1193

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using the Junos XML Protocol on page 304](#)
- [Requesting All Configuration Objects of a Specific Type Using the Junos XML Protocol on page 305](#)
- [Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol on page 310](#)

Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to request output that shows only the identifier for each configuration object of a specific type in a hierarchy, a client application emits a `<get-configuration>` tag element that encloses the tag elements representing all levels of the configuration hierarchy from the root (represented by the `<configuration>` tag element) down to the immediate parent level for the object type. An empty tag represents the requested object type, and the `recurse="false"` attribute is included. The entire request is enclosed in an `<rpc>` tag element.

To request the identifier for all objects of a specified type, the client application includes only the `recurse="false"` attribute:

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object type -->
      <object-type recurse="false"/>
      <!-- closing tags for each parent of the object type -->
    </configuration>
  </get-configuration>
</rpc>
```

To request the identifier for a specified number of objects, the client application combines the `recurse="false"` attribute with the `count` and `start` attributes discussed in [“Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol” on page 307](#):

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object type -->
      <object-type recurse="false" count="count" start="index"/>
      <!-- closing tags for each parent of the object type -->
    </configuration>
  </get-configuration>
</rpc>
```

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested objects in `<configuration>` and `<rpc-reply>` tag elements. If the application has requested a specified number of objects, the `junos:position` and `junos:total` attributes are included in the opening tag for each object, as described in [“Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol” on page 307](#).

In the following example output, the identifier tag element is called `<name>`. (For information about the attributes in the opening `<configuration>` tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).)

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the object type -->
    <first-object [junos:position="index1" junos:total="total1"]>
      <name>identifier-for-first-object</name>
    </first-object>
    <second-object [junos:position="index2" junos:total="total1"]>
      <name>identifier-for-second-object</name>
```

```

        </second-object>
        <!-- additional instances of the object -->
        <!-- closing tags for each parent of the object type -->
    </configuration>
</rpc-reply>

```

The **junos:position** and **junos:total** attributes do not appear if the client requests formatted ASCII output by including the **format="text"** attribute or if the client requests JSON-formatted output by including the **format="json"** attribute in the **<get-configuration>** tag element (as described in [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#)).

To specify the source of the output (candidate or active configuration), the application can include attributes in the opening **<get-configuration>** tag, its opening **<junoscript>** tag, or both. For more information, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-configuration>** tag element. For more information, see [“Requesting Multiple Configuration Elements Using the Junos XML Protocol” on page 317](#).

The following example shows how to request the identifier for each interface configured at the **[edit interfaces]** hierarchy level. The output is from the candidate configuration and is tagged with Junos XML tag elements (the default).

Client Application	Junos XML Protocol Server
<pre> <rpc> <get-configuration> <configuration> <interfaces> <interface recurse="false"/> </interfaces> </configuration> </get-configuration> </rpc> </pre>	<pre> <rpc-reply xmlns:junos="URL"> <configuration junos:changed-seconds='seconds' \ junos:changed-localtime='timestamp"> <interfaces> <interface> <name>fe-0/0/0</name> </interface> <interface> <name>fxp0</name> </interface> <interface> <name>lo0</name> </interface> </interfaces> </configuration> </rpc-reply> </pre>

T1194

Related Documentation • [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)

- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using the Junos XML Protocol on page 304](#)
- [Requesting All Configuration Objects of a Specific Type Using the Junos XML Protocol on page 305](#)
- [Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol on page 307](#)

Requesting a Single Configuration Object Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to request complete information about a single configuration object, a client application emits the **<get-configuration>** tag element that encloses the tag elements representing all levels of the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to the immediate parent level for the object.

To represent the requested object, the application emits only the container tag element and each of its identifier tag elements, complete with identifier value, for the object. For objects with a single identifier, the **<name>** tag element can always be used, even if the actual identifier tag element has a different name. The actual name is also valid. For objects with multiple identifiers, the actual names of the identifier tag elements must be used. To verify the name of each of the identifiers for a configuration object, see the *Junos XML API Configuration Developer Reference*. The entire request is enclosed in an **<rpc>** tag element:

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the object -->
      <object>
        <name>identifier</name>
      </object>
      <!-- closing tags for each parent of the object -->
    </configuration>
  </get-configuration>
</rpc>
```

When the client application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested object in **<configuration>** and **<rpc-reply>** tag elements. For information about the attributes in the opening **<configuration>** tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the object -->
    <object>
      <!-- child tag elements of the object -->
    </object>
    <!-- closing tags for each parent of the object -->
  </configuration>
```



```
</rpc-reply>
```

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII or JSON or an indicator for identifiers), the application can include attributes in the opening `<get-configuration>` tag, its opening `<junoscript>` tag, or both. For more information, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#) and [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#).

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same `<get-configuration>` tag element. For more information, see [“Requesting Multiple Configuration Elements Using the Junos XML Protocol” on page 317](#).

The following example shows how to request the contents of one multicasting scope called `local`, which is at the `[edit routing-options multicast]` hierarchy level. To specify the desired object, the client application emits the `<name>local</name>` identifier tag element as the innermost tag element. The output is from the candidate configuration and is tagged with Junos XML tag elements (the default).

Client Application

```
<rpc>
  <get-configuration>
    <configuration>
      <routing-options>
        <multicast>
          <scope>
            <name>local</name>
          </scope>
        </multicast>
      </routing-options>
    </configuration>
  </get-configuration>
</rpc>
```

Junos XML Protocol Server

```
<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds='seconds' \
    junos:changed-localtime='timestamp">
    <routing-options>
      <multicast>
        <scope>
          <name>local</name>
          <prefix>239.255.0.0/16</prefix>
          <interface>ip-f/p/0</interface>
        </scope>
      </multicast>
    </routing-options>
  </configuration>
</rpc-reply>
```

T1195

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)

- [Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using the Junos XML Protocol on page 304](#)
- [Requesting All Configuration Objects of a Specific Type Using the Junos XML Protocol on page 305](#)
- [Requesting Subsets of Configuration Objects Using Regular Expressions on page 314](#)
- [Requesting Multiple Configuration Elements Using the Junos XML Protocol on page 317](#)

Requesting Subsets of Configuration Objects Using Regular Expressions

In a Junos XML protocol session with a device running Junos OS, to request information about only those instances of a configuration object type that have a specified set of characters in their identifier names, a client application includes the **matching** attribute with a regular expression that matches the identifier name. For example, the application can request information about just the SONET/SDH interfaces at the **[edit interfaces]** hierarchy level by specifying the characters so- at the start of the regular expression.

Using the **matching** attribute enables the application to represent the objects to return in a form similar to the XML Path Language (XPath) representation, which is described in *XML Path Language (XPath) Version 1.0*, available from the World Wide Web Consortium (W3C) at <http://www.w3.org/TR/xpath>. In an XPath representation, an object and its parent levels are an ordered series of tag element names separated by forward slashes. The angle brackets around tag element names are omitted, and the opening tag is used to represent the entire tag element. For example, the following XPath:

```
configuration/system/radius-server/name
```

is equivalent to the following tagged representation:

```
<configuration>
  <system>
    <radius-server>
      <name/>
    </radius-server>
  </system>
</configuration>
```

The application includes the **matching** attribute in the empty tag that represents a parent level for the object type. As with all requests for configuration information, the client emits a **<get-configuration>** tag element that encloses the tag elements representing all levels of the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to the level at which the **matching** attribute is included. The entire request is enclosed in an **<rpc>** tag element:

```
<rpc>
  <get-configuration>
    <configuration>
      <!-- opening tags for each parent of the level -->
      <level matching="matching-expression"/>
      <!-- closing tags for each parent of the level -->
    </configuration>
  </get-configuration>
</rpc>
```

In the value for the **matching** attribute, each level in the XPath-like representation can be either a full level name or a regular expression that matches the identifier name of one or more instances of an object type:

```
object-type[name='regular-expression']"
```

The regular expression uses the notation defined in POSIX Standard 1003.2 for extended (modern) UNIX regular expressions. Explaining regular expression syntax is beyond the scope of this document, but [Table 7 on page 315](#) specifies which character or characters are matched by some of the regular expression operators that can be used in the expression. In the descriptions, the term *term* refers to either a single alphanumeric character or a set of characters enclosed in square brackets, parentheses, or braces.



NOTE: The **matching** attribute is not case-sensitive.

Table 7: Regular Expression Operators for the matching Attribute

Operator	Matches
. (period)	One instance of any character except the space.
* (asterisk)	Zero or more instances of the immediately preceding term.
+ (plus sign)	One or more instances of the immediately preceding term.
? (question mark)	Zero or one instance of the immediately preceding term.
(pipe)	One of the terms that appear on either side of the pipe operator.
^ (caret)	The start of a line, when the caret appears outside square brackets. One instance of any character that does not follow it within square brackets, when the caret is the first character inside square brackets.
\$ (dollar sign)	The end of a line.
[] (paired square brackets)	One instance of one of the enclosed alphanumeric characters. To indicate a range of characters, use a hyphen (-) to separate the beginning and ending characters of the range. For example, [a-z0-9] matches any letter or number.
() (paired parentheses)	One instance of the evaluated value of the enclosed term. Parentheses are used to indicate the order of evaluation in the regular expression.

When the application requests Junos XML-tagged output (the default), the Junos XML protocol server returns the requested object in `<configuration>` and `<rpc-reply>` tag elements. For information about the attributes in the opening `<configuration>` tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <configuration attributes>
    <!-- opening tags for each parent of the parent level -->
    <parent-level>
      <first-matching-object>
        <!-- child tag elements for the first object -->
      </first-matching-object>
      <second-matching-object>
        <!-- child tag elements for the second object -->
      </second-matching-object>
      <!-- additional instances of the object -->
    </parent-level>
    <!-- closing tags for each parent of the object type -->
  </configuration>
</rpc-reply>
```

The application can combine one or more of the **count**, **start**, and **recurse** attributes along with the **matching** attribute, to limit the set of possible matches to a specific range of objects, to request only identifiers, or both. For more information about those attributes, see [“Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol” on page 307](#) and [“Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol” on page 310](#).

To specify the source of the output (candidate or active configuration) and request special formatting of the output (for example, formatted ASCII or JSON or an indicator for identifiers), the application can include attributes in the opening `<get-configuration>` tag, its opening `<junoscript>` tag, or both. For more information, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#) and [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#).

The application can request additional configuration elements of the same or other types in the same `<get-configuration>` tag element by including the appropriate tag elements. For more information, see [“Requesting Multiple Configuration Elements Using the Junos XML Protocol” on page 317](#).

The following example shows how to request just the identifier for the first two SONET/SDH interfaces configured at the `[edit interfaces]` hierarchy level.

Client Application Junos XML Protocol Server

```

<rpc>
  <get-configuration>
    <configuration>
      <interfaces matching="interface[name='so-.*']" count="2" recurse="false">
    </configuration>
  </get-configuration>
</rpc>

<rpc-reply xmlns:junos="URL">
  <configuration junos:changed-seconds='seconds' \
    junos:changed-localtime='timestamp">
    <interfaces>
      <interface junos:position="41" junos:total="65">
        <name>so-0/0/0</name>
      </interface>
      <interface junos:position="42" junos:total="65">
        <name>so-0/0/1</name>
      </interface>
    </interfaces>
  </configuration>
</rpc-reply>

```

T1196

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol on page 310](#)
- [Requesting a Single Configuration Object Using the Junos XML Protocol on page 312](#)
- [Requesting Multiple Configuration Elements Using the Junos XML Protocol on page 317](#)

Requesting Multiple Configuration Elements Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, a client application can request multiple configuration elements of the same type or different types within a **<get-configuration>** tag element. The request includes only one **<configuration>** tag element (the Junos XML protocol server returns an error if there is more than one).

If two requested objects have the same parent hierarchy level, the client can either include both requests within one parent tag element, or repeat the parent tag element for each request. For example, at the **[edit system]** hierarchy level, the client can request the list of configured services and the identifier tag element for RADIUS servers in either of the following two ways:

```

<!-- both requests in one parent tag element -->
<rpc>
  <get-configuration>
    <configuration>
      <system>
        <services/>
        <radius-server>

```

```
        <name/>
      </radius-server>
    </system>
  </configuration>
</get-configuration>
</rpc>

<!-- separate parent tag element for each request -->
<rpc>
  <get-configuration>
    <configuration>
      <system>
        <services/>
      </system>
      <system>
        <radius-server>
          <name/>
        </radius-server>
      </system>
    </configuration>
  </get-configuration>
</rpc>
```

The client can combine requests for any of the types of information discussed in the following sections:

- [Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using the Junos XML Protocol on page 304](#)
- [Requesting All Configuration Objects of a Specific Type Using the Junos XML Protocol on page 305](#)
- [Requesting a Specific Number of Configuration Objects Using the Junos XML Protocol on page 307](#)
- [Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol on page 310](#)
- [Requesting a Single Configuration Object Using the Junos XML Protocol on page 312](#)
- [Requesting Subsets of Configuration Objects Using Regular Expressions on page 314](#)

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Requesting Identifiers for Configuration Objects of a Specific Type Using the Junos XML Protocol on page 310](#)
- [Requesting a Single Configuration Object Using the Junos XML Protocol on page 312](#)
- [Requesting Subsets of Configuration Objects Using Regular Expressions on page 314](#)

Retrieving a Previous (Rollback) Configuration Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to request a previously committed (rollback) configuration, a client application emits the Junos XML `<get-rollback-information>` tag element and its child `<rollback>` tag element in an `<rpc>` tag element. This operation is equivalent to the `show system rollback` operational mode command. The `<rollback>` tag element specifies the index number of the previous configuration to display; its value can be from 0 (zero, for the most recently committed configuration) through 49.

To request Junos XML-tagged output, the application either includes the `<format>` tag element with the value `xml` or omits the `<format>` tag element (Junos XML tag elements are the default):

```
<rpc>
  <get-rollback-information>
    <rollback>index-number</rollback>
  </get-rollback-information>
</rpc>
```

The Junos XML protocol server encloses its response in `<rpc-reply>`, `<rollback-information>`, and `<configuration>` tag elements. The `<load-success/>` tag is a side effect of the implementation and does not affect the results. For information about the attributes in the opening `<configuration>` tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <rollback-information>
    <load-success/>
    <configuration attributes>
      <!-- tag elements for the complete previous configuration -->
    </configuration>
  </rollback-information>
</rpc-reply>
```

To request formatted ASCII output, the application includes the `<format>` tag element with the value `text`.

```
<rpc>
  <get-rollback-information>
    <rollback>index-number</rollback>
    <format>text</format>
  </get-rollback-information>
</rpc>
```

The Junos XML protocol server encloses its response in `<rpc-reply>`, `<rollback-information>`, `<configuration-information>`, and `<configuration-output>` tag elements. For more information about the formatted ASCII notation used in Junos OS configuration statements, see [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#).

```
<rpc-reply xmlns:junos="URL">
  <rollback-information>
    <load-success/>
    <configuration-information>
      <configuration-output>
```

```
        <!-- formatted ASCII text for the complete previous
configuration -->
    </configuration-output>
</configuration-information>
</rollback-information>
</rpc-reply>
```

Starting in Junos OS Release 16.1, to request a previously committed (rollback) configuration in JSON format, the application includes the **<format>** tag element with the value **json** in the **<get-rollback-information>** element. Prior to Junos OS Release 16.1, JSON-formatted data was requested by including the **format="json"** attribute in the opening **<get-rollback-information>** tag.

```
<rpc>
  <get-rollback-information>
    <rollback>index-number</rollback>
    <format>json</format>
  </get-rollback-information>
</rpc>
```

When you use the **format="json"** attribute to specify the format, the Junos XML protocol server encloses its response in an **<rpc-reply>** element, the field name for the top-level JSON member is **"rollback-information"**, and the emitted configuration data uses an older implementation for serialization. When you use the **<format>json</format>** element to request JSON-formatted data, the Junos XML protocol server encloses its response in **<rpc-reply>**, **<rollback-information>**, **<configuration-information>**, and **<json-output>** tag elements, the field name for the top-level JSON member is **"configuration"**, and the emitted configuration data uses a newer implementation for serialization.

```
<rpc-reply xmlns:junos="URL">
  <rollback-information>
    <load-success/>
    <configuration-information>
      <json-output>
        <!-- JSON data for the complete previous configuration -->
      </json-output>
    </configuration-information>
  </rollback-information>
</rpc-reply>
```

The following example shows how to request Junos XML-tagged output for the rollback configuration that has an index of 2. In actual output, the *JUNOS-version* variable is replaced by a value such as 18.1R1 for the initial version of Junos OS Release 18.1.

Client Application	Junos XML Protocol Server
<pre> <rpc> <get-rollback-information> <rollback>2</rollback> </get-rollback-information> </rpc> </pre>	<pre> <rpc-reply xmlns:junos="URL"> <rollback-information> <load-success/> <configuration junos:changed-seconds='seconds' \ junos:changed-localtime='timestamp'> <version>JUNOS-version</version> <system> <host-name>big-router</host-name> <!-- other children of <system> - -> </system> <!-- other children of <configuration> - -> </configuration> </rollback-information> </rpc-reply> </pre>

T1197

Release History Table

Release	Description
16.1	Starting in Junos OS Release 16.1, to request a previously committed (rollback) configuration in JSON format, the application includes the <format> tag element with the value json in the <get-rollback-information> element.

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Retrieving the Rescue Configuration Using the Junos XML Protocol on page 321](#)
- [Comparing the Active or Candidate Configuration to a Prior Version Using the Junos XML Protocol on page 324](#)
- [Comparing Two Previous \(Rollback\) Configurations Using the Junos XML Protocol on page 325](#)

Retrieving the Rescue Configuration Using the Junos XML Protocol

The rescue configuration is a configuration saved in case it is necessary to restore a valid, nondefault configuration. (To create a rescue configuration, use the Junos XML **<request-save-rescue-configuration>** tag element in a Junos XML protocol session or the **request system configuration rescue save** operational mode command in the CLI. For more information, see the *Junos XML API Operational Developer Reference* or the [CLI Explorer](#).)

In a Junos XML protocol session with a device running Junos OS, a client application requests the rescue configuration by emitting the Junos XML **<get-rescue-information>**

tag element in an `<rpc>` element. This operation is equivalent to the `show system configuration rescue` operational mode command.

To request Junos XML-tagged output, the application either includes the `<format>` tag element with the value `xml` or omits the `<format>` tag element (Junos XML output is the default):

```
<rpc>
  <get-rescue-information/>
</rpc>
```

The Junos XML protocol server encloses its response in `<rpc-reply>`, `<rescue-information>`, and `<configuration>` tag elements. The `<load-success/>` tag is a side effect of the implementation and does not affect the results. For information about the attributes in the opening `<configuration>` tag, see [“Specifying the Source for Configuration Information Requests in a Junos XML Protocol Session” on page 272](#).

```
<rpc-reply xmlns:junos="URL">
  <rescue-information>
    <load-success/>
    <configuration attributes>
      <!-- tag elements representing the rescue configuration -->
    </configuration>
  </rescue-information>
</rpc-reply>
```

To request formatted ASCII output, the application includes the `<format>` tag element with the value `text`.

```
<rpc>
  <get-rescue-information>
    <format>text</format>
  </get-rescue-information>
</rpc>
```

The Junos XML protocol server encloses its response in `<rpc-reply>`, `<rescue-information>`, `<configuration-information>`, and `<configuration-output>` tag elements. For more information about the formatted ASCII notation used in Junos OS configuration statements, see [“Specifying the Output Format for Configuration Data in a Junos XML Protocol Session” on page 276](#).

```
<rpc-reply xmlns:junos="URL">
  <rescue-information>
    <load-success/>
    <configuration-information>
      <configuration-output>
        <!-- formatted ASCII text representing the rescue configuration -->
      </configuration-output>
    </configuration-information>
  </rescue-information>
</rpc-reply>
```

Starting in Junos OS Release 16.1, to request the rescue configuration in JSON format, the application includes the `<format>` tag element with the value `json` in the `<get-rescue-information>` element. Prior to Junos OS Release 16.1, JSON-formatted data was requested by including the `format="json"` attribute in the opening `<get-rescue-information>` tag.

```
<rpc>
```

```

<get-rescue-information>
  <format>json</format>
</get-rescue-information>
</rpc>

```

When you use the **format="json"** attribute to specify the format, the Junos XML protocol server encloses its response in an **<rpc-reply>** element, the field name for the top-level JSON member is **"rescue-information"**, and the emitted configuration data uses an older implementation for serialization. When you use the **<format>json</format>** element to request JSON-formatted data, the Junos XML protocol server encloses its response in **<rpc-reply>**, **<rescue-information>**, **<configuration-information>**, and **<json-output>** tag elements, the field name for the top-level JSON member is **"configuration"**, and the emitted configuration data uses a newer implementation for serialization.

```

<rpc-reply xmlns:junos="URL">
  <rescue-information>
    <load-success/>
    <configuration-information>
      <json-output>
        {
          "configuration" : {
            <!-- JSON data representing the rescue configuration -->
          }
        }
      </json-output>
    </configuration-information>
  </rescue-information>
</rpc-reply>

```

Release History Table

Release	Description
16.1	Starting in Junos OS Release 16.1, to request the rescue configuration in JSON format, the application includes the <format> tag element with the value json in the <get-rescue-information> element.

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Retrieving a Previous \(Rollback\) Configuration Using the Junos XML Protocol on page 319](#)
- [Comparing the Active or Candidate Configuration to a Prior Version Using the Junos XML Protocol on page 324](#)
- [Comparing Two Previous \(Rollback\) Configurations Using the Junos XML Protocol on page 325](#)

Comparing the Active or Candidate Configuration to a Prior Version Using the Junos XML Protocol

In the CLI, when you want to compare the active or candidate configuration to a previously committed configuration, you use the **compare** command. In operational mode, you compare the active configuration to a prior version using the **show configuration | compare rollback *rollback-number*** command. In configuration mode, you compare the candidate configuration to a previously committed configuration using the **show | compare rollback *rollback-number*** command.

The **compare rollback *rollback-number*** command compares the selected configuration with a previously committed configuration and displays the differences between the two. The *rollback-number* for the most recently saved configuration is 0, and the oldest saved configuration is 49.

In a Junos XML protocol session with a device running Junos OS, to request that the server display the differences between the active or candidate configuration and a previously committed configuration, a client application includes the **compare** and **rollback** attributes in the **<get-configuration/>** tag or opening **<get-configuration>** tag. It encloses the request in an **<rpc>** tag element:

```
<rpc>
  <get-configuration compare="rollback" rollback="[0-49]" format="text"/>

  <!-- OR -->

  <get-configuration compare="rollback" rollback="[0-49]" format="text">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
```

The client application can include the **database** attribute to specify whether to compare the active or candidate configuration to the previously committed configuration. If the **database** attribute is omitted, the candidate configuration is used. If the **rollback="rollback-number"** attribute is not included, rollback configuration number 0 is used for comparison.

By default, the **<get-configuration>** operation returns Junos XML-tagged output. However, when the **compare** attribute is included, the **<get-configuration>** operation returns the output formatted as ASCII text even if the you omit the **format="text"** attribute or try to specify a different format.

The comparison output is enclosed in the **<configuration-information>** and **<configuration-output>** tags. The output uses the following conventions to specify the differences between configurations:

- Statements that are only in the active or candidate configuration are prefixed with a plus sign (+).
- Statements that are only in the comparison file are prefixed with a minus sign (–).
- Statements that are unchanged are prefixed with a single blank space ().

- Related Documentation**
- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
 - [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
 - [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
 - [Retrieving a Previous \(Rollback\) Configuration Using the Junos XML Protocol on page 319](#)
 - [Retrieving the Rescue Configuration Using the Junos XML Protocol on page 321](#)
 - [Comparing Two Previous \(Rollback\) Configurations Using the Junos XML Protocol on page 325](#)

Comparing Two Previous (Rollback) Configurations Using the Junos XML Protocol

In a Junos XML protocol session with a device running Junos OS, to compare the contents of two previously committed (rollback) configurations, a client application emits the Junos XML `<get-rollback-information>` tag element and its child `<rollback>` and `<compare>` tag elements in an `<rpc>` tag element. This operation is equivalent to the `show system rollback` operational mode command with the `compare` option.

The `<rollback>` tag element specifies the index number of the configuration that is the basis for comparison. The `<compare>` tag element specifies the index number of the configuration to compare with the base configuration. Valid values in both tag elements range from 0 (zero, for the most recently committed configuration) through 49:

```
<rpc>
  <get-rollback-information>
    <rollback>index-number</rollback>
    <compare>index-number</compare>
  </get-rollback-information>
</rpc>
```



NOTE: The output corresponds more logically to the chronological order of changes if the older configuration (the one with the higher index number) is the base configuration. Its index number is enclosed in the `<rollback>` tag element, and the index of the more recent configuration is enclosed in the `<compare>` tag element.

The Junos XML protocol server encloses its response in `<rpc-reply>`, `<rollback-information>`, `<configuration-information>`, and `<configuration-output>` tag elements. The `<load-success/>` tag is a side effect of the implementation and does not affect the results.

The information in the `<configuration-output>` tag element is formatted ASCII text and includes a banner line (such as `[edit interfaces]`) for each hierarchy level at which the two configurations differ. Each line between banner lines begins with either a plus sign (+) or a minus sign (–). The plus sign indicates that adding the statement to the base configuration results in the second configuration, whereas a minus sign means that removing the statement from the base configuration results in the second configuration.

```
<rpc-reply xmlns:junos="URL">
  <rollback-information>
```

```

    <load-success/>
    <configuration-information>
      <configuration-output>
        <!-- formatted ASCII text representing the changes -->
      </configuration-output>
    </configuration-information>
  </rollback-information>
</rpc-reply>

```

The following example shows how to request a comparison of the rollback configurations that have indexes of 20 and 4.

Client Application

```

<rpc>
  <get-rollback-information>
    <rollback>20</rollback>
    <compare>4</compare>
  </get-rollback-information>
</rpc>

```

Junos XML Protocol Server

```

<rpc-reply xmlns:junos="URL">
  <rollback-information>
    <load-success/>
    <configuration-information>
      <configuration-output>
        [edit interfaces]
        - ge-0/2/0 {
        -   stacked-vlan-tagging;
        -   mac 00.01.02.03.04.05;
        -   gigether-options {
        -     loopback;
        -   }
        - }
        [edit]
        + services {
        +   l2tp {
        +     tunnel-group 12 {
        +       local-gateway;
        +     }
        +   }
        + }
      </configuration-output>
    </configuration-information>
  </rollback-information>
</rpc-reply>

```

T1170

Related Documentation

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Specifying the Scope of Configuration Data to Return in a Junos XML Protocol Session on page 302](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)
- [Retrieving a Previous \(Rollback\) Configuration Using the Junos XML Protocol on page 319](#)
- [Retrieving the Rescue Configuration Using the Junos XML Protocol on page 321](#)
- [Comparing the Active or Candidate Configuration to a Prior Version Using the Junos XML Protocol on page 324](#)

Requesting an XML Schema for the Configuration Hierarchy Using the Junos XML Protocol

The schema represents all configuration elements available in the version of the Junos OS that is running on a device. (To determine the Junos OS version, emit the **<get-software-information>** operational request tag, which is documented in the *Junos XML API Operational Developer Reference*.)

Client applications can use the schema to validate the configuration on a device, or simply to learn which configuration statements are available in the version of the Junos OS running on the device. The schema does not indicate which elements are actually configured, or even that an element can be configured on that type of device (some configuration statements are available only on certain device types). To request the set of currently configured elements and their settings, emit the **<get-configuration>** tag element instead, as described in “Requesting Configuration Data Using the Junos XML Protocol” on page 270.

Explaining the structure and notational conventions of the XML Schema language is beyond the scope of this document. For information, see *XML Schema Part 0: Primer*, available from the World Wide Web Consortium (W3C) at <http://www.w3.org/TR/xmlschema-0/>. The primer provides a basic introduction and lists the formal specifications where you can find detailed information.

For further information, see the following sections:

- Requesting an XML Schema for the Configuration Hierarchy on page 327
- Creating the junos.xsd File on page 328
- Example: Requesting an XML Schema on page 328

Requesting an XML Schema for the Configuration Hierarchy

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

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

The Junos XML protocol server encloses the XML schema in **<rpc-reply>** and **<xsd:schema>** tags:

```
<rpc-reply xmlns:junos="URL">
  <xsd:schema>
    <!-- tag elements for the Junos schema -->
  </xsd:schema>
</rpc-reply>
```

Creating the junos.xsd File

Most of the tag elements defined in the schema returned in the `<xsd:schema>` tag belong to the default namespace for Junos OS configuration elements. However, at least one tag, `<junos:comment>`, belongs to a different namespace:

`http://xml.juniper.net/junos/Junos-version/junos`. By XML convention, a schema describes only one namespace, so schema validators need to import information about any additional namespaces before they can process the schema.

Starting in Junos OS Release 6.4, the `<xsd:import>` tag element is enclosed in the `<xsd:schema>` tag element and references the file `junos.xsd`, which contains the required information about the `junos` namespace. For example, the following `<xsd:import>` tag element specifies the file for Junos OS Release 18.1R1 (and appears on two lines for legibility only):

```
<xsd:import schemaLocation="junos.xsd" \
  namespace="http://xml.juniper.net/junos/18.1R1/junos"/>
```

To enable the schema validator to interpret the `<xsd:import>` tag, you must manually create a file called `junos.xsd` in the directory where you place the `.xsd` file that contains the complete Junos configuration schema. Include the following text in the file. Do not use line breaks in the list of attributes in the opening `<xsd:schema>` tag. Line breaks appear in the following example for legibility only. For the `Junos-version` variable, substitute the release number of the Junos OS running on the device (for example, 18.1R1 for the first release of Junos OS 18.1).

```
<?xml version="1.0" encoding="us-ascii"?>
<xsd:schema elementFormDefault="qualified" \
  attributeFormDefault="unqualified" \
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" \
  targetNamespace="http://xml.juniper.net/junos/Junos-version/junos">
  <xsd:element name="comment" type="xsd:string"/>
</xsd:schema>
```



NOTE: Schema validators might not be able to process the schema if they cannot locate or open the `junos.xsd` file.

Whenever you change the version of Junos OS running on the device, remember to update the `Junos-version` variable in the `junos.xsd` file to match.

Example: Requesting an XML Schema

The following examples show how to request the Junos OS configuration schema. In the Junos XML protocol server's response, the first `<xsd:element>` statement defines the `<undocumented>` Junos XML tag element, which can be enclosed in most other container tag elements defined in the schema (container tag elements are defined as `<xsd:complexType>`).

The attributes in the opening tags of the Junos XML protocol server's response appear on multiple lines for legibility only. The Junos XML protocol server does not insert newline characters within tags or tag elements. Also, in actual output the `JUNOS-version` variable is replaced by a value such as 18.1R1 for the initial version of Junos OS Release 18.1.

Client Application Junos XML Protocol Server

```

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

<rpc-reply xmlns:junos="URL">
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" \
    elementFormDefault="qualified">
    <xsd:import schemaLocation="junos.xsd" \
      namespace="http://xml.juniper.net/junos/ Junos-version/junos"/>
    <xsd:element name="undocumented">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:any namespace="##any" processContents="skip"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:complexType name="hostname">
      <xsd:simpleContent>
        <xsd:extension base="xsd:string"/>
      </xsd:simpleContent>
    </xsd:complexType>
  </xsd:schema>
</rpc-reply>

```

T1177

Another `<xsd:element>` statement near the beginning of the schema defines the Junos XML `<configuration>` tag element. It encloses the `<xsd:element>` statement that defines the `<system>` tag element, which corresponds to the **[edit system]** hierarchy level. The statements corresponding to other hierarchy levels are omitted for brevity.

Client Application Junos XML Protocol Server

```

</xsd:element>
<xsd:element name="configuration">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="undocumented"/>
        <xsd:element ref="comment"/>
        <xsd:element name="system" minOccurs="0">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="undocumented"/>
                <xsd:element ref="comment"/>
                <!-- child elements of <system> -->
              </xsd:choice>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <!-- definitions for other hierarchy levels -->
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
</rpc-reply>

```

T1178

- Related Documentation**
- [Understanding the Request Procedure in a Junos XML Protocol Session on page 54](#)
 - [Requesting Operational Information Using the Junos XML Protocol on page 255](#)

- [Requesting Configuration Data Using the Junos XML Protocol on page 270](#)
- [Requesting the Complete Configuration Using the Junos XML Protocol on page 302](#)

PART 5

Junos XML Protocol Utilities

- [Junos XML Protocol Perl Client on page 333](#)
- [Developing Junos XML Protocol Perl Client Applications on page 341](#)
- [Developing Junos XML Protocol C Client Applications on page 369](#)

Junos XML Protocol Perl Client

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Downloading the Junos XML Protocol Perl Client and Prerequisites Package on page 334](#)
- [Installing the Junos XML Protocol Perl Client and Prerequisites Package on page 335](#)

Understanding the Junos XML Protocol Perl Distribution and Sample Scripts

Juniper Networks provides a Perl module **JUNOS::Device** to help you more quickly and easily develop custom Perl scripts for configuring and monitoring switches, routers, and security devices running Junos OS. The module implements a **JUNOS::Device** object that client applications can use to communicate with the Junos XML protocol server on a device running Junos OS. The Perl distribution includes several sample Perl scripts, which illustrate how to use the module in scripts that perform various functions.

The Junos XML protocol Perl distribution uses the same directory structure for Perl modules as the [Comprehensive Perl Archive Network](#). This includes a **lib** directory for the **JUNOS** module and its supporting files, and an **examples** directory for the sample scripts.

Client applications use the **JUNOS::Device** object to communicate with a Junos XML protocol server. The library contains several modules, but client applications directly invoke only the **JUNOS::Device** object. All of the sample scripts use this object.

The sample scripts illustrate how to perform the following functions:

- **diagnose_bgp.pl**—Illustrates how to write scripts to monitor device status and diagnose problems. The sample script extracts and displays information about a device's unestablished Border Gateway Protocol (BGP) peers from the full set of BGP configuration data. The script is in the **examples/diagnose_bgp** directory in the Junos XML protocol Perl distribution.
- **get_chassis_inventory.pl**—Illustrates how to use a predefined query to request information from a device. The sample script invokes the **get_chassis_inventory** method with the **detail** option to request the same information as the Junos XML **get-chassis-inventorydetail/get-chassis-inventory** tag sequence and the command-line interface (CLI) operational mode command **show chassis hardware detail**. The script is in the **examples/get_chassis_inventory** directory in the Junos XML protocol Perl distribution.

- **load_configuration.pl**—Illustrates how to change a device configuration by loading a file that contains configuration data formatted with Junos XML tag elements. The distribution includes two sample configuration files, **set_login_class_bar.xml** and **set_login_user_foo.xml**; however, you can specify a different configuration file on the command line. The script is in the **examples/load_configuration** directory in the Junos XML protocol Perl distribution.

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

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

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

Related Documentation

- [Supported Access Protocols for Junos XML Protocol Sessions on page 52](#)
- [Downloading the Junos XML Protocol Perl Client and Prerequisites Package on page 334](#)
- [Installing the Junos XML Protocol Perl Client and Prerequisites Package on page 335](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Connecting to the Junos XML Protocol Server Using Perl Client Applications on page 344](#)
- [Mapping Junos OS Commands to Perl Methods on page 351](#)

Downloading the Junos XML Protocol Perl Client and Prerequisites Package

To download the compressed tar archives that contain the Junos XML protocol Perl client distribution and the prerequisites package, perform the following steps:

1. Access the download page for Junos XML management protocol at <https://www.juniper.net/support/downloads/?p=junoscript>.
2. Select the appropriate software release.

3. Select the Software tab.
4. Click the links to download the client distribution and the prerequisites package that support the appropriate access protocols. Customers in the United States and Canada can download the packages that support all access protocols including SSH, SSL, clear-text and Telnet protocols (the domestic package). Customers in other countries can download the packages that support only the clear-text and Telnet protocols (the export package).



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

Optionally, download the packages containing the document type definitions (DTDs) and the XML Schema language representation of the Junos OS configuration hierarchy:

1. Access the download page for the Junos XML API at <https://www.juniper.net/support/downloads/?p=junosxml>.
2. Select the appropriate software release.
3. Select the Software tab.
4. Click the links to download the desired packages.

Related Documentation

- [Supported Access Protocols for Junos XML Protocol Sessions on page 52](#)
- [Installing the Junos XML Protocol Perl Client and Prerequisites Package on page 335](#)
- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)

Installing the Junos XML Protocol Perl Client and Prerequisites Package

To install the Junos XML protocol Perl client and the prerequisites package, perform the following procedures:

- [Verifying Installation and Version of Perl on page 336](#)
- [Extracting the Junos XML Protocol Perl Client and Sample Scripts on page 336](#)
- [Extracting and Installing the Junos XML Protocol Perl Client Prerequisites Package on page 337](#)
- [Installing the Junos XML Protocol Perl Client on page 338](#)

Verifying Installation and Version of Perl

Perl must be installed on your system before you install the Junos XML protocol Perl client prerequisites package or client software. The Junos XML protocol Perl client requires Perl version 5.0004 or later. To confirm whether Perl is installed on your system and to determine which version of Perl is currently running, issue the following commands:

```
% which perl
% perl -v
```

If the issued output indicates that Perl is not installed or the version is older than the required version, you must download and install Perl version 5.0004 or later in order to use the Junos XML protocol Perl client. The Perl source packages are located at:

<http://www.cpan.org/src/>.

After installing a suitable version of Perl, extract the Junos XML protocol Perl client, extract and install the prerequisites package, and then install the Junos XML protocol Perl client application.

Extracting the Junos XML Protocol Perl Client and Sample Scripts

To uncompress and extract the contents of the compressed tar archive that contains the Junos XML protocol Perl client and sample scripts, perform the following steps:

1. Create the directory where you want to store the Junos XML protocol Perl client application and sample scripts, and move the downloaded client application file into this directory. Then make this directory the working directory:

```
% mkdir parent-directory
% mv junoscript-perl-release-type.tar.gz parent-directory
% cd parent-directory
```

2. Issue the following command to uncompress and extract the contents of the Junos XML protocol Perl client package:

- On FreeBSD and Linux systems:

```
% tar xzf junoscript-perl-release-type.tar.gz
```

- On Solaris systems:

```
% gzip -dc junoscript-perl-release-type.tar.gz | tar xf
```

where *release* is the release code, for example 18.1R1.1, and *type* is **domestic** or **export**.

Step 2 creates a directory called **junoscript-perl-*release*** and extracts the contents of the tar archive to it. For example, a typical filename for the compressed tar archive is **junoscript-perl-9.5R1.8-domestic.tar.gz**. Extracting the contents of this archive creates the directory **junoscript-perl-9.5R1.8** directly under ***parent-directory*** and places the application files and sample scripts into this new directory.

The `junoscript-perl-release/README` file contains instructions for extracting and installing the Perl prerequisite modules, creating a `Makefile`, and installing and testing the `JUNOS::Device` module.

Extracting and Installing the Junos XML Protocol Perl Client Prerequisites Package

The prerequisites package consists of C libraries, executables, and Perl modules. It must be installed on the client machine in order for the Junos XML protocol Perl client and the included examples to work correctly. To uncompress and extract the contents of the compressed tar archive containing the prerequisite files, perform the following steps:

1. Move the downloaded prerequisites package into the `parent-directory/junoscript-perl-release/` directory that was created in [“Extracting the Junos XML Protocol Perl Client and Sample Scripts” on page 336](#). The compressed tar archive containing the prerequisite files must be uncompressed, unpacked, and installed in this directory.
2. Issue the following command to uncompress and extract the contents of the package:
 - On FreeBSD and Linux systems:


```
% tar xzf junoscript-perl-prereqs-release-type.tar.gz
```
 - On Solaris systems:


```
% gzip -dc junoscript-perl-prereqs-release-type.tar.gz | tar xf
```

where **release** is the release code, for example 18.1R1.1, and **type** is **domestic** or **export**. This command creates a directory called `prereqs/` and extracts the contents of the tar archive to it.

By default, the prerequisite Perl modules are installed in the standard directory. The standard directory is normally `/usr/local/lib/`. You need root privileges to access the standard directory. You can opt to install the modules in a private directory.

- To install the required modules in the standard directory:
 1. Go to the `junoscript-perl-release/` directory where you extracted the contents of the prerequisites package.
 2. Issue the following command:


```
% perl install-prereqs.pl -used_by example -force
```

where the `-used_by example` option is invoked to install only modules used by a specific example, and the `-force` option installs the module even if an older version exists or if the `make test` command fails.
- To install the required modules in a private directory:
 1. Set the `PERL5LIB`, `MANPATH`, and `PATH` environment variables.


```
% setenv PERL5LIB private-directory-path
% setenv MANPATH "$MANPATH:$PERL5LIB/./man"
% setenv PATH "$PATH:$PERL5LIB/./bin"
```

For sh, ksh, and bash shells, **\$PERL5LIB** can be set with **EXPORT**
PERL5LIB=private-directory-path

2. Go to the **junoscript-perl-release** directory where you extracted the contents of the prerequisites package.
3. Issue the following command:

```
% perl install-prereqs.pl -used_by example -install_directory $PERL5LIB -force
```

where the **-used_by example** option is invoked to install only modules used by a specific example, and the **-force** option installs the module even if an older version exists or if the **make test** command fails. The **-install_directory \$PERL5LIB** option installs the prerequisite Perl modules in the private directory that you specified in Step 1.

Installation log files are written to **junoscript-perl-release/tmp/output/**. After installation, you can view any missing dependencies by issuing the following command:

```
% perl required-mod.pl
```

This command lists the modules that still require installation.

Installing the Junos XML Protocol Perl Client

After installing the prerequisites package as detailed in [“Extracting and Installing the Junos XML Protocol Perl Client Prerequisites Package” on page 337](#), install the Junos XML protocol Perl client software. Go to the **junoscript-perl-release/** directory that was created in [“Extracting the Junos XML Protocol Perl Client and Sample Scripts” on page 336](#). Perform the following steps to install the client software:

1. Create the makefile:
 - To install the Perl client in the standard directory (generally **/usr/local/lib**):

```
% perl Makefile.PL
```

```
Checking if your kit is complete...
Looks good
Writing Makefile for junoscript-perl
```

- To install the Perl client in a private directory:

Make sure that the **PERL5LIB**, **MANPATH**, and **PATH** environment variables are set as detailed in [“Extracting and Installing the Junos XML Protocol Perl Client Prerequisites Package” on page 337](#). Then create the makefile:

```
% perl Makefile.PL LIB=$PERL5LIB INSTALLMAN3DIR=$PERL5LIB/../man/man3
```

2. Test and install the application:

```
% make
% make test
% make install
```

The Junos XML protocol Perl client application is installed and ready for use. For information about the **JUNOS::Device** object and a list of valid queries, consult the man page by invoking the **man** command for the **JUNOS::Device** object:

% man JUNOS::Device

The sample scripts reside in the **junoscript-perl-release/examples/** directory. You can review and run these examples to acquire some familiarity with the client before writing your own applications.

**Related
Documentation**

- [Supported Access Protocols for Junos XML Protocol Sessions on page 52](#)
- [Downloading the Junos XML Protocol Perl Client and Prerequisites Package on page 334](#)
- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)

CHAPTER 17

Developing Junos XML Protocol Perl Client Applications

- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Importing Perl Modules and Declaring Constants in Junos XML Protocol Perl Client Applications on page 342](#)
- [Connecting to the Junos XML Protocol Server Using Perl Client Applications on page 344](#)
- [Collecting Parameters Interactively in Junos XML Protocol Perl Client Applications on page 347](#)
- [Converting Disallowed Characters in Junos XML Protocol Perl Client Applications on page 349](#)
- [Mapping Junos OS Commands to Perl Methods on page 351](#)
- [Submitting a Request to the Junos XML Protocol Server in Perl Client Applications on page 352](#)
- [Requesting an Inventory of Hardware Components Using Junos XML Protocol Perl Client Applications on page 356](#)
- [Example: Changing the Configuration Using Junos XML Protocol Perl Client Applications on page 356](#)
- [Parsing and Formatting Junos XML Protocol Server Responses in Perl Client Applications on page 361](#)
- [Closing the Connection to the Junos XML Protocol Server in Perl Client Applications on page 367](#)

Writing Junos XML Protocol Perl Client Applications

The following outline lists the basic tasks involved in writing a Junos XML protocol Perl client application that requests operational or configuration information from the Junos XML protocol server or loads configuration information onto a device running Junos OS. Each task provides a link to detailed information about performing the task. The tasks reference the sample scripts included with the Junos XML Protocol Perl distribution as examples.

1. Import Perl Modules and Declare Constants— and [“Importing Perl Modules and Declaring Constants in Junos XML Protocol Perl Client Applications” on page 342](#)
2. Connect to the Junos XML protocol Server—[“Connecting to the Junos XML Protocol Server Using Perl Client Applications” on page 344](#) and [“Collecting Parameters Interactively in Junos XML Protocol Perl Client Applications” on page 347](#)
3. Submit Requests to the Junos XML protocol Server—[“Submitting a Request to the Junos XML Protocol Server in Perl Client Applications” on page 352](#)
4. Parse and Format the Response from the Junos XML protocol Server—[“Parsing and Formatting Junos XML Protocol Server Responses in Perl Client Applications” on page 361](#)
5. Close the Connection to the Junos XML protocol Server—[“Closing the Connection to the Junos XML Protocol Server in Perl Client Applications” on page 367](#)

Related Documentation

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Converting Disallowed Characters in Junos XML Protocol Perl Client Applications on page 349](#)
- [Mapping Junos OS Commands to Perl Methods on page 351](#)
- [Requesting an Inventory of Hardware Components Using Junos XML Protocol Perl Client Applications on page 356](#)
- [Example: Changing the Configuration Using Junos XML Protocol Perl Client Applications on page 356](#)

Importing Perl Modules and Declaring Constants in Junos XML Protocol Perl Client Applications

When creating a Junos XML protocol Perl client application, include the following statements at the start of the application. The `use JUNOS::Device;` statement imports the functions provided by the `JUNOS::Device` object, which the application uses to connect to the Junos XML protocol server on a device running Junos OS. The `use strict` statement

provides error checking and enforces Perl coding practices such as declaration of variables before use.

```
use JUNOS::Device;
use strict;
```

Include statements to import other Perl modules as appropriate for your application. For example, several of the sample scripts included in the Junos XML protocol Perl distribution import the following standard Perl modules, which include functions that handle input from the command line:

- **File::Basename**—Includes functions for processing filenames.
- **Getopt::Std**—Includes functions for reading in keyed options from the command line.
- **Term::ReadKey**—Includes functions for controlling terminal modes, for example suppressing onscreen echo of a typed string such as a password.

If the application uses constants, declare their values at this point. For example, the sample **diagnose_bgp.pl** script includes the following statements to declare constants for formatting output:

```
use constant OUTPUT_FORMAT => "%-20s%-8s%-8s%-11s%-14s%\n";
use constant OUTPUT_TITLE =>
    "\n===== BGP PROBLEM SUMMARY =====\n\n";
use constant OUTPUT_ENDING =>
    "\n===== \n\n";
```

The **load_configuration.pl** sample script includes the following statements to declare constants for reporting return codes and the status of the configuration database:

```
use constant REPORT_SUCCESS => 1;
use constant REPORT_FAILURE => 0;
use constant STATE_CONNECTED => 1;
use constant STATE_LOCKED => 2;
use constant STATE_CONFIG_LOADED => 3;
```

Related Documentation

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Connecting to the Junos XML Protocol Server Using Perl Client Applications on page 344](#)
- [Converting Disallowed Characters in Junos XML Protocol Perl Client Applications on page 349](#)

Connecting to the Junos XML Protocol Server Using Perl Client Applications

The following sections explain how to use the **JUNOS::Device** object in a Perl client application to connect to the Junos XML protocol server on a device running Junos OS:

- [Satisfying Protocol Prerequisites on page 344](#)
- [Group Requests on page 344](#)
- [Obtaining and Recording Parameters Required by the JUNOS::Device Object on page 344](#)
- [Obtaining Application-Specific Parameters on page 345](#)
- [Establishing the Connection on page 346](#)

Satisfying Protocol Prerequisites

The Junos XML protocol server supports several access protocols. For each connection to the Junos XML protocol server on a device running Junos OS, the application must specify the protocol it is using. Using SSH or Secure Sockets Layer (SSL) is recommended because they provide greater security by encrypting all information before transmission across the network.

Before your application can run, you must satisfy the prerequisites for the access protocol it uses. For some protocols this involves activating configuration statements on the device, creating encryption keys, or installing additional software on the device running Junos OS or the machine where the application runs. For instructions, see [“Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server” on page 57](#).

Group Requests

Establishing a connection to the Junos XML protocol server on a device running Junos OS is one of the more time-intensive and resource-intensive functions performed by an application. If the application sends multiple requests to a routing platform, it makes sense to send all of them within the context of one connection. If your application sends the same requests to multiple devices, you can structure the script to iterate through either the set of devices or the set of requests. Keep in mind, however, that your application can effectively send only one request to one Junos XML protocol server at a time. This is because the **JUNOS::Device** object does not return control to the application until it receives the closing **</rpc-reply>** tag that represents the end of the Junos XML protocol server's response to the current request.

Obtaining and Recording Parameters Required by the JUNOS::Device Object

The **JUNOS::Device** object takes the following parameters, specified as keys in a Perl hash:

- **access**—(Required) The access protocol to use when communicating with the Junos XML protocol server. Acceptable values are "telnet", "ssh", "clear-text", and "ssl". Before the application runs, satisfy any protocol-specific prerequisites.
- **hostname**—(Required) The name of the device to which to connect. For best results, specify either a fully qualified hostname or an IP address.

- **login**—(Required) The username under which to establish the connection to the Junos XML protocol server and issue requests. The username must already exist on the specified device and have the permission bits necessary for making the requests invoked by the application.
- **password**—(Required) The password corresponding to the username.
- **junos_key**—(Optional) When requesting configuration data using the `get_configuration()` method, set the value of the parameter to 1 to include the `junos:key="key"` attribute in the output of XML-formatted configuration data. The default value of the parameter is 0. The `junos:key="key"` attribute indicates that a child configuration tag element is the identifier for its parent tag element. The attribute does not appear in formatted ASCII output.

The sample scripts record the parameters in a Perl hash called `%deviceinfo`, declared as follows:

```
my %deviceinfo = (
    access => $access,
    login => $login,
    password => $password,
    hostname => $hostname,
    junos_key => $junos_key,
);
```

The sample scripts included in the Junos XML Protocol Perl client distribution obtain the parameters from options entered on the command line by a user. For more information about collecting parameter values interactively, see [“Collecting Parameters Interactively in Junos XML Protocol Perl Client Applications” on page 347](#). Your application can also obtain values for the parameters from a file or database, or you can hardcode one or more of the parameters into the application code if they are constant.

Obtaining Application-Specific Parameters

In addition to the parameters required by the `JUNOS::Device` object, applications might need to define other parameters, such as the name of the file to which to write the data returned by the Junos XML protocol server in response to a request, or the name of the Extensible Stylesheet Transformation Language (XSLT) file to use for transforming the data.

As with the parameters required by the `JUNOS::Device` object, the client application can hardcode the values in the application code, obtain them from a file, or obtain them interactively. The sample scripts obtain values for these parameters from command-line options in the same manner as they obtain the parameters required by the `JUNOS::Device` object (discussed in [“Obtaining and Recording Parameters Required by the JUNOS::Device Object” on page 344](#)). Several examples follow.

The following line enables a debugging trace if the user includes the `-d` command-line option. It invokes the `JUNOS::Trace::init` routine defined in the `JUNOS::Trace` module, which is imported with the `JUNOS::Device` object.

```
JUNOS::Trace::init(1) if $opt{d};
```

The following line sets the **\$outputfile** variable to the value specified by the **-o** command-line option. It names the local file to which the Junos XML protocol server's response is written. If the **-o** option is not provided, the variable is set to the empty string.

```
my $outputfile = $opt{o} || " " ;
```

The following code from the **diagnose_bgp.pl** script defines which XSLT file to use to transform the Junos XML protocol server's response. The first line sets the **\$xslfile** variable to the value specified by the **-x** command-line option. If the option is not provided, the script uses the **text.xsl** file supplied with the script, which transforms the data to ASCII text. The **if** statement verifies that the specified XSLT file exists; the script terminates if it does not.

```
# Retrieve the XSLT file, default is parsed by perl
my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
    die "ERROR: XSLT file $xslfile does not exist";
}
```

The following code from the **load_configuration.pl** script defines whether to merge, replace, update, or overwrite the new configuration data into the configuration database. The first two lines set the **\$load_action** variable to the value of the **-a** command-line option, or to the default value **merge** if the option is not provided. If the specified value does not match one of the valid actions defined by the **VALID_ACTIONS** constant, the script invokes the **output_usage** subroutine to print the usage message.

```
# The default action for load_configuration is 'merge'
my $load_action = "merge";
$load_action = $opt{a} if $opt{a};
use constant VALID_ACTIONS => "merge|replace|override";
output_usage() unless (VALID_ACTIONS =~ /$load_action/);
```

Establishing the Connection

After obtaining values for the parameters required for the **JUNOS::Device** object, each sample script records them in the **%deviceinfo** hash.

```
my %deviceinfo = (
    access => $access,
    login => $login,
    password => $password,
    hostname => $hostname,
    junos_key => $junos_key,
);
```

The script then invokes the Junos XML protocol-specific **new** subroutine to create a **JUNOS::Device** object and establish a connection to the specified routing, switching, or security platform. If the connection attempt fails (as tested by the **ref** operator), the script exits.

```
my $jnx = new JUNOS::Device(%deviceinfo);
unless ( ref $jnx ) {
    die "ERROR: $deviceinfo{hostname}: failed to connect.\n";
}
```

Related Documentation

- [Supported Access Protocols for Junos XML Protocol Sessions on page 52](#)
- [Satisfying the Prerequisites for Establishing a Connection to the Junos XML Protocol Server on page 57](#)

- [Collecting Parameters Interactively in Junos XML Protocol Perl Client Applications on page 347](#)
- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Closing the Connection to the Junos XML Protocol Server in Perl Client Applications on page 367](#)

Collecting Parameters Interactively in Junos XML Protocol Perl Client Applications

In a Junos XML protocol Perl client application, a script can interactively obtain the parameters required by the **JUNOS::Device** object from command-line. The Junos XML Protocol Perl distribution includes several sample Perl scripts that perform various functions on devices running Junos OS. Each sample script obtains the parameters required by the **JUNOS::Device** object from command-line options provided by the user who invokes the script. The script records the options in a Perl hash called **%opt**, using the **getopts** function defined in the **Getopt::Std** Perl module to read the options from the command line. (Scripts used in production environments probably do not obtain parameters interactively, so this section is important mostly for understanding the sample scripts.)

In the following example from the **get_chassis_inventory.pl** sample script, the first parameter to the **getopts** function defines the acceptable options, which vary depending on the application. A colon after the option letter indicates that it takes an argument.

The second parameter, **\%opt**, specifies that the values are recorded in the **%opt** hash. If the user does not provide at least one option, provides an invalid option, or provides the **-h** option, the script invokes the **output_usage** subroutine, which prints a usage message to the screen.

```
my %opt;
getopts('l:p:dx:m:o:h', \%opt) || output_usage();
output_usage() if $opt{h};
```

The following code defines the **output_usage** subroutine for the **get_chassis_inventory.pl** sample script. The contents of the **my \$usage** definition and the **Where** and **Options** sections are specific to the script, and differ for each application.

```
sub output_usage
{
    my $usage = "Usage: $0 [options] <target>

    Where:
        <target>    The hostname of the target device.

    Options:

        -l <login>    A login name accepted by the target device.
        -p <password> The password for the login name.
        -m <access>   Access method. It can be clear-text, ssl, ssh or telnet.

                    Default: telnet.
```

```
-x <format>      The name of the XSL file to display the response.
                  Default: xsl/chassis_inventory_csv.xsl
-o <filename>    File to which to write output, instead of standard
output.
-d              Turn on debugging.\n\n";

    die $usage;
}
```

The `get_chassis_inventory.pl` script includes the following code to obtain values from the command line for the four parameters required by the `JUNOS::Device` object. A detailed discussion of the various functional units follows the complete code sample.

```
# Get the hostname
my $hostname = shift || output_usage();

# Get the access method
my $access = $opt{m} || "telnet";
use constant VALID_ACCESSSES => "telnet|ssh|clear-text|ssl";
output_usage() unless (VALID_ACCESSSES =~ /$access/);

# Check for login name. If not provided, prompt for it
my $login = "";
if ($opt{l}) {
    $login = $opt{l};
} else {
    print STDERR "login: ";
    $login = ReadLine 0;
    chomp $login;
}

# Check for password. If not provided, prompt for it
my $password = "";
if ($opt{p}) {
    $password = $opt{p};
} else {
    print STDERR "password: ";
    ReadMode 'noecho';
    $password = ReadLine 0;
    chomp $password;
    ReadMode 'normal';
    print STDERR "\n";
}
```

In the first line of the preceding code sample, the script uses the Perl `shift` function to read the hostname from the end of the command line. If the hostname is missing, the script invokes the `output_usage` subroutine to print the usage message, which specifies that a hostname is required.

```
my $hostname = shift || output_usage();
```

The script next determines which access protocol to use, setting the `$access` variable to the value of the `-m` command-line option or to the value "telnet" if the `-m` option is not provided. If the specified value does not match one of the values defined by the `VALID_ACCESSSES` constant, the script invokes the `output_usage` subroutine to print the usage message.

```
my $access = $opt{m} || "telnet";
use constant VALID_ACCESSSES => "telnet|ssh|clear-text|ssl";
output_usage() unless (VALID_ACCESSSES =~ /$access/);
```

The script then determines the username, setting the `$login` variable to the value of the `-l` command-line option. If the option is not provided, the script prompts for it and uses the `ReadLine` function (defined in the standard Perl `Term::ReadKey` module) to read it from the command line.

```
my $login = "";
if ($opt{l}) {
    $login = $opt{l};
} else {
    print STDERR "login: ";
    $login = ReadLine 0;
    chomp $login;
}
```

The script finally determines the password for the username, setting the `$password` variable to the value of the `-p` command-line option. If the option is not provided, the script prompts for it. It uses the `ReadMode` function (defined in the standard Perl `Term::ReadKey` module) twice: first to prevent the password from echoing visibly on the screen and then to return the shell to normal (echo) mode after it reads the password.

```
my $password = "";
if ($opt{p}) {
    $password = $opt{p};
} else {
    print STDERR "password: ";
    ReadMode 'noecho';
    $password = ReadLine 0;
    chomp $password;
    ReadMode 'normal';
    print STDERR "\n";
}
```

Related Documentation

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Connecting to the Junos XML Protocol Server Using Perl Client Applications on page 344](#)
- [Closing the Connection to the Junos XML Protocol Server in Perl Client Applications on page 367](#)

Converting Disallowed Characters in Junos XML Protocol Perl Client Applications

Junos XML protocol Perl client applications that handle configuration data usually accept and output the data either as Junos XML tag elements or as formatted ASCII statements like those used in the Junos OS CLI. As described in [“XML and Junos XML Management Protocol Conventions Overview” on page 11](#), certain characters cannot appear in their regular form in an XML document. These characters include the apostrophe (`'`), the ampersand (`&`), the greater-than (`>`) and less-than (`<`) symbols, and the quotation mark (`"`). Because these characters might appear in formatted ASCII configuration statements, the script must convert the characters to the corresponding predefined entity references.

The Junos XML Protocol Perl distribution includes several sample Perl scripts that perform various functions on devices running Junos OS. The sample scripts include code to convert all disallowed characters to predefined entity references. The `load_configuration.pl` sample script uses the `get_escaped_text` subroutine to substitute predefined entity references for disallowed characters (the `get_configuration.pl` script includes similar code). The script first defines the mappings between the disallowed characters and predefined entity references, and sets the variable `$char_class` to a regular expression that contains all of the entity references, as follows:

```
my %escape_symbols = (
    qq(") => '&quot;';
    qq(>) => '&gt;';
    qq(<) => '&lt;';
    qq(') => '&apos;';
    qq(&) => '&amp;';
);
my $char_class = join ("|", map { "($_)" } keys %escape_symbols);
```

The following code defines the `get_escaped_text` subroutine for the `load_configuration.pl` script. A detailed discussion of the subsections in the routine follows the complete code sample.

```
sub get_escaped_text
{
    my $input_file = shift;
    my $input_string = "";

    open(FH, $input_file) or return undef;

    while(<FH>) {
        my $line = $_;
        $line =~ s/<configuration-text>//g;
        $line =~ s/<\/configuration-text>//g;
        $line =~ s/($char_class)/$escape_symbols{$1}/ge;
        $input_string .= $line;
    }

    return "<configuration-text>$input_string</configuration-text>";
}
```

The first subsection of the preceding code sample reads in a file containing formatted ASCII configuration statements.

```
sub get_escaped_text
{
    my $input_file = shift;
    my $input_string = "";

    open(FH, $input_file) or return undef;
```

In the next subsection, the subroutine temporarily discards the lines that contain the opening `<get-configuration>` and closing `</get-configuration>` tags, then replaces the disallowed characters on each remaining line with predefined entity references and appends the line to the `$input_string` variable:

```
while(<FH>) {
    my $line = $_;
    $line =~ s/<configuration-text>//g;
    $line =~ s/<\/configuration-text>//g;
    $line =~ s/($char_class)/$escape_symbols{$1}/ge;
```

```

    $input_string .= $line;
}

```

The subroutine concludes by replacing the opening `<get-configuration>` and closing `</get-configuration>` tags, and returning the converted set of statements:

```

    return "<configuration-text>$input_string</configuration-text>";
}

```

**Related
Documentation**

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Submitting a Request to the Junos XML Protocol Server in Perl Client Applications on page 352](#)
- [Example: Changing the Configuration Using Junos XML Protocol Perl Client Applications on page 356](#)

Mapping Junos OS Commands to Perl Methods

Juniper Networks provides Perl modules to help you more quickly and easily develop custom Perl scripts for configuring and monitoring switches, routers, and security devices running Junos OS. The modules implement an object that client applications can use to communicate with the NETCONF or Junos XML protocol server on a device running Junos OS. The NETCONF and Junos XML protocol Perl distributions include sample Perl scripts, which illustrate how to use the modules in scripts that perform various functions.

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

To derive the Perl method name from the request tag element name, replace each hyphen in the tag element name with an underscore and remove the enclosing angle brackets from the tag element name. For example, the `<get-bgp-group-information>` tag element maps to the `get_bgp_group_information` Perl method.

The sample scripts included with the Perl distributions invoke only a small number of the Junos XML Perl methods available in Junos OS. For a list of all of the request tag elements available in the current version of the Junos OS, see the chapter in the *Junos XML API Operational Developer Reference* that maps Junos XML request tag elements to CLI commands and Perl methods. For information about optional and required attributes for a particular Perl method, see the entry for the corresponding Junos XML request tag element in the chapters titled “Summary of Operational Request Tag Elements” in the *Junos XML API Operational Developer Reference*.

**Related
Documentation**

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)

- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Submitting a Request to the Junos XML Protocol Server in Perl Client Applications on page 352](#)
- [Requesting an Inventory of Hardware Components Using Junos XML Protocol Perl Client Applications on page 356](#)
- [Example: Changing the Configuration Using Junos XML Protocol Perl Client Applications on page 356](#)
- [Parsing and Formatting Junos XML Protocol Server Responses in Perl Client Applications on page 361](#)

Submitting a Request to the Junos XML Protocol Server in Perl Client Applications

In a Junos XML Protocol Perl client application, after establishing a connection to a Junos XML protocol server, the client application can submit one or more requests by invoking the Perl methods that are supported in the version of the Junos XML protocol and Junos XML API used by the application.

- Each version of software supports a set of methods that correspond to CLI operational mode commands. For a list of the operational methods supported in the current version, see [“Mapping Junos OS Commands to Perl Methods” on page 351](#) and the files stored in the `lib/JUNOS/release` directory of the Junos XML protocol Perl distribution (*release* is the Junos OS version code, such as 18.1R1 for the initial version of Junos OS Release 18.1). The files have names in the format `package_methods.pl`, where *package* is a software package.
- The set of methods that correspond to operations on configuration objects is defined in the `lib/JUNOS/Methods.pm` file in the Junos XML protocol Perl distribution.

See the following sections for more information:

- [Providing Method Options or Attributes on page 352](#)
- [Submitting a Request on page 354](#)

Providing Method Options or Attributes

Many Perl methods have one or more options or attributes. The following list describes the notation used to define a method's options in the `lib/JUNOS/Methods.pm` and `lib/JUNOS/release/package_methods.pl` files, and the notation that an application uses when invoking the method:

- A method without options is defined as `$NO_ARGS`, as in the following entry for the `get_system_uptime_information` method:

```
## Method : <get-system-uptime-information>
## Returns: <system-uptime-information>
## Command: "show system uptime"
get_system_uptime_information => $NO_ARGS,
```


To invoke a method without options, follow the method name with an empty set of parentheses as in the following example:

```
$jnx->get_system_uptime_information( );
```

- A fixed-form option is defined as type **\$TOGGLE**. In the following example, the **get_software_information** method takes two fixed-form options, **brief** and **detail**:

```
## Method : <get-software-information>
## Returns: <software-information>
## Command: "show version"
get_software_information =>
    brief => $TOGGLE,
    detail => $TOGGLE,
},
```

To include a fixed-form option when invoking a method, set it to the value **1** (one) as in the following example:

```
$jnx->get_software_information(brief => 1);
```

- An option with a variable value is defined as type **\$STRING**. In the following example, the **get_cos_drop_profile_information** method takes the **profile_name** argument:

```
## Method : <get-cos-drop-profile-information>
## Returns: <cos-drop-profile-information>
## Command: "show class-of-service drop-profile"
get_cos_drop_profile_information => {
    profile_name => $STRING,
},
```

To include a variable value when invoking a method, enclose the value in single quotes as in the following example:

```
$jnx->get_cos_drop_profile_information(profile_name => 'user-drop-profile');
```

- An attribute is defined as type **\$ATTRIBUTE**. In the following example, the **load_configuration** method takes the **rollback** attribute:

```
load_configuration => {
    rollback => $ATTRIBUTE
},
```

To include a numerical attribute value when invoking a method, set it to the appropriate value. The following example rolls the candidate configuration back to the previous configuration that has an index of 2:

```
$jnx->load_configuration(rollback => 2);
```

To include a string attribute value when invoking a method, enclose the value in single quotes as in the following example:

```
$jnx->get_configuration(format => 'text');
```

- A set of configuration statements or corresponding tag elements is defined as type **\$DOM**. In the following example, the **get_configuration** method takes a set of configuration statements (along with two attributes):

```
get_configuration => {
    configuration => $DOM,
    format => $ATTRIBUTE,
    database => $ATTRIBUTE,
},
```

To include a set of configuration statements when invoking a method, provide a parsed set of statements or tag elements. The following example refers to a set of Junos XML configuration tag elements in the **config-input.xml** file. For a more detailed discussion about modifying the configuration, see [“Example: Changing the Configuration Using Junos XML Protocol Perl Client Applications” on page 356](#).

```
my $parser = new XML::DOM::Parser;
$jnx->load_configuration(
    format => 'xml',
    action => 'merge',
    configuration => $parser->parsefile(config-input.xml)
);
```

A method can have a combination of fixed-form options, options with variable values, attributes, and a set of configuration statements. For example, the **get_forwarding_table_information** method has four fixed-form options and five options with variable values:

```
## Method : <get-forwarding-table-information>
## Returns: <forwarding-table-information>
## Command: "show route forwarding-table"
get_forwarding_table_information => {
    detail => $TOGGLE,
    extensive => $TOGGLE,
    multicast => $TOGGLE,
    family => $STRING,
    vpn => $STRING,
    summary => $TOGGLE,
    matching => $STRING,
    destination => $STRING,
    label => $STRING,
},
```

Submitting a Request

The following code is the recommended way to send a request to the Junos XML protocol server and shows how to handle error conditions. The **\$jnx** variable is defined to be a **JUNOS::Device** object, as discussed in [“Connecting to the Junos XML Protocol Server Using Perl Client Applications” on page 344](#). A detailed discussion of the functional subsections follows the complete code sample.

```
my %arguments = ( );
%arguments = ( argument1 => value1 ,
    argument2 => value2 , ...);
    argument3 => value3 ,
    ...);

my $res = $jnx-> method (%args);

unless ( ref $res ) {
    $jnx->request_end_session( );
    $jnx->disconnect( );
    print "ERROR: Could not send request to $hostname\n";
}

my $err = $res->getFirstError( );
if ($err) {
    $jnx->request_end_session( );
    $jnx->disconnect();
}
```

```
        print "ERROR: Error for $hostname: " . $err->{message} . "\n";
    }
}
```

The first subsection of the preceding code sample creates a hash called **%arguments** to define values for a method's options or attributes. For each argument, the application uses the notation described in [“Providing Method Options or Attributes” on page 352](#).

```
my %arguments = ( );
%arguments = ( argument1 => value1 ,
              argument2 => value2 , ...);
              argument3 => value3 ,
              ...);
```

The application then invokes the method, defining the **\$res** variable to point to the **JUNOS::Response** object that the Junos XML protocol server returns in response to the request (the object is defined in the **lib/JUNOS/Response.pm** file in the Junos XML protocol Perl distribution):

```
my $res = $jnx-> method (%args);
```

If the attempt to send the request failed, the application prints an error message and closes the connection:

```
unless ( ref $res ) {
    $jnx->request_end_session();
    $jnx->disconnect();
    print "ERROR: Could not send request to $hostname\n";
}
```

If there was an error in the Junos XML protocol server's response, the application prints an error message and closes the connection. The **getFirstError** function is defined in the **JUNOS::Response** module (**lib/JUNOS/Response.pm**) in the Junos XML protocol Perl distribution.

```
my $err = $res->getFirstError( );
if ($err) {
    $jnx->request_end_session( );
    $jnx->disconnect( );
    print "ERROR: Error for $hostname: " . $err->{message} . "\n";
}
```

Related Documentation

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Mapping Junos OS Commands to Perl Methods on page 351](#)
- [Requesting an Inventory of Hardware Components Using Junos XML Protocol Perl Client Applications on page 356](#)
- [Example: Changing the Configuration Using Junos XML Protocol Perl Client Applications on page 356](#)
- [Parsing and Formatting Junos XML Protocol Server Responses in Perl Client Applications on page 361](#)

Requesting an Inventory of Hardware Components Using Junos XML Protocol Perl Client Applications

The Junos XML protocol Perl distribution includes several sample Perl scripts that perform various functions on devices running Junos OS. The **get_chassis_inventory.pl** script retrieves and displays a detailed inventory of the hardware components installed in a routing, switching, or security platform. The output is equivalent to issuing the **show chassis hardware detail** operational mode command in the Junos OS command-line interface (CLI). This topic describes the portion of the script that executes the query.

After establishing a connection to the Junos XML protocol server, the script sends the **get_chassis_inventory** request and includes the **detail** argument.

```
my $query = "get_chassis_inventory";
my %queryargs = ( detail => 1 );
```

The script sends the query and assigns the results to the **\$res** variable. It performs two tests on the results, and prints an error message if it cannot send the request or if errors occurred when executing it. If no errors occur, the script uses XSLT to transform the results. For more information, see [“Parsing and Formatting Junos XML Protocol Server Responses in Perl Client Applications” on page 361](#).

```
# send the command and receive a XML::DOM object
my $res = $jnx->$query( %queryargs );
unless ( ref $res ) {
    die "ERROR: $deviceinfo{hostname}: failed to execute command $query.\n";
}
# Check and see if there were any errors in executing the command.
my $err = $res->getFirstError( );
if ($err) {
    print STDERR "ERROR: $deviceinfo{'hostname'} - ", $err->{message}, "\n";
} else {
    # Now do the transformation using XSLT
    ... code that uses XSLT to process results ...
}
```

Related Documentation

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Mapping Junos OS Commands to Perl Methods on page 351](#)
- [Submitting a Request to the Junos XML Protocol Server in Perl Client Applications on page 352](#)
- [Example: Changing the Configuration Using Junos XML Protocol Perl Client Applications on page 356](#)
- [Parsing and Formatting Junos XML Protocol Server Responses in Perl Client Applications on page 361](#)

Example: Changing the Configuration Using Junos XML Protocol Perl Client Applications

The Junos XML protocol Perl distribution includes several sample Perl scripts that perform various functions on devices running Junos OS. The **load_configuration.pl** script locks,

modifies, uploads, and commits the configuration on a device. It uses the basic structure for sending requests described in [“Submitting a Request to the Junos XML Protocol Server in Perl Client Applications” on page 352](#) but also defines a **graceful_shutdown** subroutine that handles errors. The following sections describe the different functions that the script performs:

- [Handling Error Conditions on page 357](#)
- [Locking the Configuration on page 358](#)
- [Reading In and Parsing the Configuration Data on page 358](#)
- [Loading the Configuration Data on page 359](#)
- [Committing the Configuration on page 360](#)

Handling Error Conditions

The **graceful_shutdown** subroutine in the **load_configuration.pl** script handles errors encountered in the Junos XML protocol session. It employs the following additional constants:

```
use constant REPORT_SUCCESS => 1;
use constant REPORT_FAILURE => 0;
use constant STATE_CONNECTED => 1;
use constant STATE_LOCKED => 2;
use constant STATE_CONFIG_LOADED => 3;
```

The first two **if** statements in the subroutine refer to the **STATE_CONFIG_LOADED** and **STATE_LOCKED** conditions, which apply specifically to loading a configuration in the **load_configuration.pl** script. The **eval** statement ensures that any errors that occur during execution of the enclosed function call are trapped so that failure of the function call does not cause the script to exit.

```
sub graceful_shutdown
{
    my ($jnx, $req, $state, $success) = @_;

    if ($state >= STATE_CONFIG_LOADED) {
        print "Rolling back configuration ...\n";
        eval {
            $jnx->load_configuration(rollback => 0);
        };
    }

    if ($state >= STATE_LOCKED) {
        print "Unlocking configuration database ...\n";
        eval {
            $jnx->unlock_configuration();
        };
    }

    if ($state >= STATE_CONNECTED) {
        print "Disconnecting from the device ...\n";
        eval {
            $jnx->request_end_session()
            $jnx->disconnect();
        };
    }
}
```

```
    if ($success) {
        die "REQUEST $req SUCCEEDED\n";
    } else {
        die "REQUEST $req FAILED\n";
    };
}
```

Locking the Configuration

The main section of the **load_configuration.pl** script begins by establishing a connection to a Junos XML protocol server. It then invokes the **lock_configuration** method to lock the configuration database. If an error occurs, the script invokes the **graceful_shutdown** subroutine described in [“Handling Error Conditions” on page 357](#).

```
print "Locking configuration database ...\n";
my $res = $jnx->lock_configuration();
my $err = $res->getFirstError();
if ($err) {
    print "ERROR: $deviceinfo{hostname}: failed to lock configuration. Reason:
    $err->{message}.\n";
    graceful_shutdown($jnx, $xmlfile, STATE_CONNECTED, REPORT_FAILURE);
}
```

Reading In and Parsing the Configuration Data

In the following code sample, the **load_configuration.pl** script reads in and parses a file that contains Junos XML configuration tag elements or ASCII-formatted statements. A detailed discussion of the functional subsections follows the complete code sample.

```
# Load the configuration from the given XML file
print "Loading configuration from $xmlfile ...\n";
if (! -f $xmlfile) {
    print "ERROR: Cannot load configuration in $xmlfile\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}

my $parser = new XML::DOM::Parser;
...

my $doc;
if ($opt{t}) {
    my $xmlstring = get_escaped_text($xmlfile);
    $doc = $parser->parsestring($xmlstring) if $xmlstring;
} else {
    $doc = $parser->parsefile($xmlfile);
}

unless ( ref $doc ) {
    print "ERROR: Cannot parse $xmlfile, check to make sure the XML data is
    well-formed\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}
```

The first subsection of the preceding code sample verifies the existence of the file containing configuration data. The name of the file was previously obtained from the command line and assigned to the **\$xmlfile** variable. If the file does not exist, the script invokes the **graceful_shutdown** subroutine:

```

print "Loading configuration from $xmlfile ...\n";
if (! -f $xmlfile) {
    print "ERROR: Cannot load configuration in $xmlfile\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}

```

If the `-t` command-line option was included when the `load_configuration.pl` script was invoked, the file referenced by the `$xmlfile` variable should contain formatted ASCII configuration statements like those returned by the CLI configuration-mode `show` command. The script invokes the `get_escaped_text` subroutine described in [“Converting Disallowed Characters in Junos XML Protocol Perl Client Applications” on page 349](#), assigning the result to the `$xmlstring` variable. The script invokes the `parsestring` function to transform the data in the file into the proper format for loading into the configuration hierarchy, and assigns the result to the `$doc` variable. The `parsestring` function is defined in the `XML::DOM::Parser` module, and the first line in the following sample code instantiates the module as an object, setting the `$parser` variable to refer to it:

```

my $parser = new XML::DOM::Parser;
...
my $doc;
if ($opt{t}) {
    my $xmlstring = get_escaped_text($xmlfile);
    $doc = $parser->parsestring($xmlstring) if $xmlstring;
}

```

If the `-t` command-line option was included when the `load_configuration.pl` script was invoked, the file referenced by the `$xmlfile` variable contains Junos XML configuration tag elements instead. In this case, the script invokes the `parsefile` function (also defined in the `XML::DOM::Parser` module) on the file.

```

} else {
    $doc = $parser->parsefile($xmlfile);
}

```

If the parser cannot transform the file, the script invokes the `graceful_shutdown` subroutine described in [“Handling Error Conditions” on page 357](#).

```

unless ( ref $doc ) {
    print "ERROR: Cannot parse $xmlfile, check to make sure the XML data is
well-formed\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}

```

Loading the Configuration Data

The script now invokes the `load_configuration` method to load the configuration changes onto the device. It places the statement inside an `eval` block to ensure that the `graceful_shutdown` subroutine is invoked if the response from the Junos XML protocol server has errors.

```

eval {
    $res = $jnx->load_configuration(
        format => $config_format,
        action => $load_action,
        configuration => $doc);
};
if ($?) {
    print "ERROR: Failed to load the configuration from $xmlfile. Reason:
    $@\n";
}

```

```

        graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
        exit(1);
    }

```

The variables used to define the method's three arguments were set at previous points in the application file:

- The **\$config_format** variable is set to "xml" unless the **-t** command-line option was included when invoking the script.

```

my $config_format = "xml";
$config_format = "text" if $opt{t};

```

- The **\$load_action** variable is set to "merge" unless the **-a** command-line option was included when invoking the script. The final two lines verify that the specified value is valid.

```

my $load_action = "merge";
$load_action = $opt{a} if $opt{a};
use constant VALID_ACTIONS => "merge|replace|override";
output_usage() unless ( $load_action =~ /VALID_ACTIONS/);

```

- The **\$doc** variable contains the output from the **parsestring** or **parsefile** function (defined in the **XML::DOM::Parser** module).

The script performs two additional checks for errors and invokes the **graceful_shutdown** subroutine in either case.

```

unless ( ref $res ) {
    print "ERROR: Failed to load the configuration from $xmlfile\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}
$error = $res->getFirstError();
if ($error) {
    print "ERROR: Failed to load the configuration. Reason: $error->{message}\n";

    graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
}

```

Committing the Configuration

If there are no errors up to this point, the script invokes the **commit_configuration** method (defined in the file **lib/JUNOS/Methods.pm** in the Junos XML protocol Perl distribution) to commit the configuration on the device and make it the active configuration.

```

print "Committing configuration from $xmlfile ...\n";
$res = $jnx->commit_configuration();
$error = $res->getFirstError();
if ($error) {
    print "ERROR: Failed to commit configuration. Reason: $error->{message}.\n";

    graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
}

```

Related Documentation

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)

- [Submitting a Request to the Junos XML Protocol Server in Perl Client Applications on page 352](#)
- [Requesting an Inventory of Hardware Components Using Junos XML Protocol Perl Client Applications on page 356](#)
- [Parsing and Formatting Junos XML Protocol Server Responses in Perl Client Applications on page 361](#)

[Parsing and Formatting Junos XML Protocol Server Responses in Perl Client Applications](#)

In a Junos XML Protocol Perl client application, as the last step in sending a request, the application verifies that there are no errors with the response from the Junos XML protocol server. It can then write the response to a file, to the screen, or both. If the response is for an operational query, the application usually uses XSLT to transform the output into a more readable format, such as HTML or formatted ASCII text. If the response consists of configuration data, the application can store it as XML (the Junos XML tag elements generated by default from the Junos XML protocol server) or transform it into formatted ASCII text.

The following sections discuss parsing and formatting options:

- [Parsing and Formatting an Operational Response on page 361](#)
- [Parsing and Outputting Configuration Data on page 363](#)

[Parsing and Formatting an Operational Response](#)

The Junos XML protocol Perl distribution includes several sample Perl scripts that perform various functions on devices running Junos OS. The following code sample from the **diagnose_bgp.pl** and **get_chassis_inventory.pl** sample scripts uses XSLT to transform an operational response from the Junos XML protocol server into a more readable format. A detailed discussion of the functional subsections follows the complete code sample.

```
# Get the name of the output file
my $outputfile = $opt{o} || "";

# Retrieve the XSLT file
my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
    die "ERROR: XSLT file $xslfile does not exist";
}

#Get the xmlfile
my $xmlfile = "$deviceinfo{hostname}.xml";
$res->printToFile($xmlfile);

my $nm = $res->translateXSLtoRelease('xmlns:lc', $xslfile, "$xslfile.tmp");

if ($nm) {
    print "Transforming $xmlfile with $xslfile...\n" if $outputfile;
    my $command = "xsltproc $nm $deviceinfo{hostname}.xml";

    $command .= "> $outputfile" if $outputfile;
    system($command);
    print "Done\n" if $outputfile;
}
```

```

        print "See $outputfile\n" if $outputfile;
    }

    else {
        print STDERR "ERROR: Invalid XSL file $xslfile\n";
    }
}

```

The first line of the preceding code sample illustrates how the scripts read the `-o` option from the command line to obtain the name of the file into which to write the results of the XSLT transformation:

```
my $outputfile = $opt{o} || "";
```

The scripts use an XSLT file to transform the output. If the `-x` command-line option is included when the script is invoked, the script uses the XSLT file specified in the option. Otherwise, it uses the default XSLT file specified in the code. The script exits if the file does not exist. The following example is from the **diagnose_bgp.pl** script:

```

my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
    die "ERROR: XSLT file $xslfile does not exist";
}

```

For examples of XSLT files, see the following directories in the Junos XML protocol Perl distribution:

- The **examples/diagnose_bgp/xsl** directory contains XSLT files for the **diagnose_bgp.pl** script: **dhtml.xsl** generates dynamic HTML, **html.xsl** generates HTML, and **text.xsl** generates ASCII text.
- The **examples/get_chassis_inventory/xsl** directory contains XSLT files for the **get_chassis_inventory.pl** script: **chassis_inventory_csv.xsl** generates a list of comma-separated values, **chassis_inventory_html.xsl** generates HTML, and **chassis_inventory_xml.xsl** generates XML.

The actual parsing operation begins by setting the variable `$xmlfile` to a filename of the form **device-name.xml** and invoking the **printToFile** function to write the Junos XML protocol server's response into the file (the **printToFile** function is defined in the **XML::DOM::Parser** module):

```

my $xmlfile = "$deviceinfo{hostname}.xml";
$res->printToFile($xmlfile);

```

The next line invokes the **translateXSLtoRelease** function (defined in the **Junos::Response** module) to alter one of the namespace definitions in the XSLT file. This is necessary because the XSLT 1.0 specification requires that every XSLT file define a specific value for each default namespace used in the data being transformed. The **xmlns** attribute in a Junos XML operational response tag element includes a code representing the Junos OS version, such as 18.1R1 for the initial version of Junos OS Release 18.1. Because the same XSLT file can be applied to operational response tag elements from devices running different versions of the Junos OS, the XSLT file cannot predefine an **xmlns** namespace value that matches all versions. The **translateXSLtoRelease** function alters the namespace definition in the XSLT file identified by the `$xslfile` variable to match the value in the Junos XML protocol server's response. It assigns the resulting XSLT file to the `$nm` variable.

```
my $nm = $res->translateXSLtoRelease('xmlns:1c', $xslfile, "$xslfile.tmp");
```

After verifying that the **translateXSLtoRelease** function succeeded, the invokes the **format_by_xslt**, which builds a command string and assigns it to the **\$command** variable. The first part of the command string invokes the **xsltproc** command and specifies the names of the XSLT and configuration data files (stored in **\$nm** and **\$deviceinfo{hostname}.xml**):

```
if ($nm) {
    print "Transforming $xmlfile with $xslfile...\n" if $outputfile;
    my $command = "xsltproc $nm $deviceinfo{hostname}.xml";
```

If the **\$outputfile** variable is defined (the file for storing the result of the XSLT transformation exists), the script appends a string to the **\$command** variable to write the results of the **xsltproc** command to the file. (If the file does not exist, the script writes the results to standard out [stdout].) The script then invokes the **system** function to execute the command string and prints status messages to stdout.

```
    $command .= "> $outputfile" if $outputfile;
    system($command);
    print "Done\n" if $outputfile;
    print "See $outputfile\n" if $outputfile;
}
```

If the **translateXSLtoRelease** function fails (the **if (\$nm)** expression evaluates to "false"), the script prints an error:

```
else {
    print STDERR "ERROR: Invalid XSL file $xslfile\n";
}
```

Parsing and Outputting Configuration Data

The **get_config.pl** script in the **examples\RDB** directory uses the **outconfig** subroutine to write the configuration data obtained from the Junos XML protocol server to a file either as Junos XML tag elements or as formatted ASCII text.

The **outconfig** subroutine takes four parameters. Three must have defined values: the directory in which to store the output file, device hostname, and the XML DOM tree (the configuration data) returned by the Junos XML protocol server. The fourth parameter indicates whether to output the configuration as formatted ASCII text, and has a null value if the requested output is Junos XML tag elements. In the following code sample, the script obtains values for the four parameters and passes them to the **outconfig** subroutine. A detailed discussion of each line follows the complete code sample.

```
my(%opt,$login,$password);

getopts('!p:dm:hit', \%opt) || output_usage();
output_usage() if $opt{h};

my $basepath = shift || output_usage;

my $hostname = shift || output_usage;

my $config = getconfig( $hostname, $jnx, $opt{t} );

outconfig( $basepath, $hostname, $config, $opt{t} );
```

In the first lines of the preceding sample code, the **get_config.pl** script uses the following statements to obtain values for the four parameters to the **outconfig** subroutine:

- If the user provides the **-t** option on the command line, the **getopts** subroutine records it in the **%opt** hash. The value keyed to **\$opt{t}** is passed as the fourth parameter to the **outconfig** subroutine. (For more information about reading options from the command line, see [“Collecting Parameters Interactively in Junos XML Protocol Perl Client Applications”](#) on page 347.)

```
getopts('l:p:dm:hit', \%opt) || output_usage();
```

- The following line reads the first element of the command line that is not an option preceded by a hyphen. It assigns the value to the **\$basepath** variable, defining the name of the directory in which to store the file containing the output from the **outconfig** subroutine. The variable value is passed as the first parameter to the **outconfig** subroutine.

```
my $basepath = shift || output_usage;
```

- The following line reads the next element on the command line. It assigns the value to the **\$hostname** variable, defining the routing, switching, or security device hostname. The variable value is passed as the second parameter to the **outconfig** subroutine.

```
my $hostname = shift || output_usage;
```

- The following line invokes the **getconfig** subroutine to obtain configuration data from the Junos XML protocol server on the specified device, assigning the resulting XML DOM tree to the **\$config** variable. The variable value is passed as the third parameter to the **outconfig** subroutine.

```
my $config = getconfig( $hostname, $jnx, $opt{t} );
```

The following code sample invokes and defines the **outconfig** subroutine. A detailed discussion of each functional subsection in the subroutine follows the complete code sample.

```
outconfig( $basepath, $hostname, $config, $opt{t} );

sub outconfig( $$$ ) {
    my $leader = shift;
    my $hostname = shift;
    my $config = shift;
    my $text_mode = shift;
    my $trailer = "xmlconfig";
    my $filename = $leader . "/" . $hostname . "." . $trailer;

    print "# storing configuration for $hostname as $filename\n";

    my $config_node;
    my $top_tag = "configuration";
    $top_tag .= "-text" if $text_mode;
    if ($config->getTagName() eq $top_tag) {
        $config_node = $config;
    } else {
        print "# unknown response component ", $config->getTagName(), "\n";
    }

    if ( $config_node && $config_node ne "" ) {
```

```

        if ( open OUTPUTFILE, ">$filename" )    {
            if (!$text_mode) {
                print OUTPUTFILE "<?xml version=\"1.0\"?>\n";
                print OUTPUTFILE $config_node->toString(), "\n";
            } else {
                my $buf = $config_node->getFirstChild()->toString();
                $buf =~ s/($char_class)/$escapes{$1}/ge;
                print OUTPUTFILE "$buf\n";
            }
            close OUTPUTFILE;
        }
    } else {
        print "ERROR: could not open output file $filename\n";
    }
}
else {
    print "ERROR: empty configuration data for $hostname\n";
}
}
}

```

The first lines of the **outconfig** subroutine read in the four parameters passed in when the subroutine is invoked, assigning each to a local variable:

```

outconfig( $basepath, $hostname, $config, $opt{t} );
sub outconfig( $$$ ) {
    my $leader = shift;
    my $hostname = shift;
    my $config = shift;
    my $text_mode = shift;

```

The subroutine constructs the name of the file to which to write the subroutine's output and assigns the name to the **\$filename** variable. The filename is constructed from the first two parameters (the directory name and hostname) and the **\$trailer** variable, resulting in a name of the form **directory-name/hostname.xmlconfig**:

```

my $trailer = "xmlconfig";
my $filename = $leader . "/" . $hostname . "." . $trailer;

print "# storing configuration for $hostname as $filename\n";

```

The subroutine checks that the first tag in the XML DOM tree correctly indicates the type of configuration data in the file. If the user included the **-t** option on the command line, the first tag should be **<configuration-text>** because the file contains formatted ASCII configuration statements; otherwise, the first tag should be **<configuration>** because the file contains Junos XML tag elements. The subroutine sets the **\$stop_tag** variable to the appropriate value depending on the value of the **\$text_mode** variable (which takes its value from **opt{t}**, passed as the fourth parameter to the subroutine). The subroutine invokes the **getTagName** function (defined in the **XML::DOM::Element** module) to retrieve the name of the first tag in the input file, and compares the name to the value of the **\$stop_tag** variable. If the comparison succeeds, the XML DOM tree is assigned to the **\$config_node** variable. Otherwise, the subroutine prints an error message because the XML DOM tree is not valid configuration data.

```

my $config_node;
my $stop_tag = "configuration";
$stop_tag .= "-text" if $text_mode;
if ($config->getTagName( ) eq $stop_tag) {
    $config_node = $config;
} else {

```

```
        print "# unknown response component ", $config->getTagName( ), "\n";
    }
}
```

The subroutine then uses several nested **if** statements. The first **if** statement verifies that the XML DOM tree exists and contains data:

```
if ( $config_node && $config_node ne "" ) {
    ... actions if XML DOM tree contains data ...
}
else {
    print "ERROR: empty configuration data for $hostname\n";
}
```

If the XML DOM tree contains data, the subroutine verifies that the output file can be opened for writing:

```
if ( open OUTPUTFILE, ">$filename" ) {
    ... actions if output file is writable ...
}
else {
    print "ERROR: could not open output file $filename\n";
}
```

If the output file can be opened for writing, the script writes the configuration data into it. If the user requested Junos XML tag elements—the user did not include the **-t** option on the command line, so the **\$text_mode** variable does not have a value—the script writes the string **<?xml version=1.0?>** as the first line in the output file, and then invokes the **toString** function (defined in the **XML::DOM** module) to write each Junos XML tag element in the XML DOM tree on a line in the output file:

```
if (!$text_mode) {
    print OUTPUTFILE "<?xml version=\"1.0\"?>\n";
    print OUTPUTFILE $config_node->toString( ), "\n";
}
```

If the user requested formatted ASCII text, the script invokes the **getFirstChild** and **toString** functions (defined in the **XML::DOM** module) to write the content of each tag on its own line in the output file. The script substitutes predefined entity references for disallowed characters (which are defined in the **%escapes** hash), writes the output to the output file, and closes the output file. (For information about defining the **%escapes** hash to contain the set of disallowed characters, see [“Converting Disallowed Characters in Junos XML Protocol Perl Client Applications” on page 349.](#))

```
} else {
    my $buf = $config_node->getFirstChild( )->toString( );
    $buf =~ s/($char_class)/$escapes{$1}/ge;
    print OUTPUTFILE "$buf\n";
}
close OUTPUTFILE;
```

Related Documentation

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Mapping Junos OS Commands to Perl Methods on page 351](#)
- [Submitting a Request to the Junos XML Protocol Server in Perl Client Applications on page 352](#)
- [Requesting an Inventory of Hardware Components Using Junos XML Protocol Perl Client Applications on page 356](#)

- [Example: Changing the Configuration Using Junos XML Protocol Perl Client Applications on page 356](#)

Closing the Connection to the Junos XML Protocol Server in Perl Client Applications

In Junos XML protocol Perl client applications, you can end the Junos XML protocol session and close the connection to the device by invoking the **request_end_session** and **disconnect** methods. Several of the sample scripts included in the Junos XML protocol Perl client distribution invoke the **request_end_session** and **disconnect** methods in standalone statements. For example:

```
$jnx->request_end_session( );  
$jnx->disconnect();
```

The **load_configuration.pl** script invokes the **graceful_shutdown** subroutine instead (for more information, see [“Handling Error Conditions” on page 357](#)).

```
graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_SUCCESS);
```

Related Documentation

- [Understanding the Junos XML Protocol Perl Distribution and Sample Scripts on page 333](#)
- [Writing Junos XML Protocol Perl Client Applications on page 342](#)
- [Connecting to the Junos XML Protocol Server Using Perl Client Applications on page 344](#)
- [Mapping Junos OS Commands to Perl Methods on page 351](#)

Developing Junos XML Protocol C Client Applications

- [Establishing a Junos XML Protocol Session Using C Client Applications on page 369](#)
- [Accessing and Editing Device Configurations Using Junos XML Protocol C Client Applications on page 370](#)

Establishing a Junos XML Protocol Session Using C Client Applications

This example illustrates how a Junos XML protocol C client application uses the SSH or Telnet protocol to establish a connection and Junos XML protocol session with a device running Junos OS. In the line that begins with the string **exec1p**, the client application invokes the **ssh** command. (Substitute the **telnet** command if appropriate.) The *routing-platform* argument to the **exec1p** routine specifies the hostname or IP address of the Junos XML protocol server device. The **junoscript** argument is the command that converts the connection to a Junos XML protocol session.

```
int ipipes[ 2 ], opipes[ 2 ];
pid_t pid;
int rc;
char buf[ BUFSIZ ];

if (pipe(ipipes) <0 || pipe(opipes) <0)
    err(1, "pipe failed");

pid = fork( );
if (pid <0)
    err(1, "fork failed");

if (pid == 0) {
    dup2(opipes[ 0 ], STDIN_FILENO);
    dup2(ipipes[ 1 ], STDOUT_FILENO);
    dup2(ipipes[ 1 ], STDERR_FILENO);
    close(ipipes[ 0 ]); /* close read end of pipe */
    close(ipipes[ 1 ]); /* close write end of pipe */
    close(opipes[ 0 ]); /* close read end of pipe */
    close(opipes[ 1 ]); /* close write end of pipe */

    exec1p("ssh", "ssh", "-x", routing-platform , "junoscript", NULL);
    err (1, "unable to execute: ssh %s junoscript," device);
}
```

```

close(ipipes[ 1 ]); /* close write end of pipe */
close(opipes[ 0 ]); /* close read end of pipe */

if (write(opipes[ 1 ], initial_handshake, strlen(initial_handshake)) <0
)
    err(1, "writing initial handshake failed");

rc=read(ipipes[ 0 ], buf, sizeof(buf));
if (rc <0)
    err(1, "read initial handshake failed");

```

- Related Documentation**
- [Accessing and Editing Device Configurations Using Junos XML Protocol C Client Applications on page 370](#)
 - [Writing Junos XML Protocol Perl Client Applications on page 342](#)

Accessing and Editing Device Configurations Using Junos XML Protocol C Client Applications

This example script presents a C client application that can be used to access, edit, and commit configurations on routers, switches, and security devices running Junos OS

```

/--Includes--//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <errno.h>
#include <libxml/parser.h>
#include <libxml/xpath.h>

/--Defines--//
#define PRINT
    //--Toggles printing of all data to and from js server--//

/--Global Variables and Initialization--//
int sockfd;
char *xmlns_start_ptr = NULL;
char *xmlns_end_ptr = NULL;
int sock_bytes, pim_output_len, igmp_output_len, count_a, count_x, count_y,
    count_z, repl_str_len, orig_len, up_to_len, remain_len, conf_chg;
struct sockaddr_in serv_addr;
struct hostent *server;
char temp_buff[1024];                //--Temporary buffer used when --//
                                     //--sending js configuration commands--//
char rcvbuffer[255];                //--Stores data from socket--//
char *pim_output_ptr = NULL;        //--Pointer for pim_output from socket--//
                                     //--buffer--//
char *igmp_output_ptr = NULL;        //--Pointer for igmp_output from socket
buffer--//
char small_buff[2048];              //--Buffer to support js communication--//
char jserver[16];                   //--Junos XML protocol server IP address--//
int jport = 3221;                   //--Junos XML protocol server port --//

```

```

char msource[16];                //--(xnm-clear-text)--//
                                //--Multicast source of group being
                                //--configured under igmp--//
char minterface[16];             //--Local multicast source interface--//
                                //--###change in igmp_xpath_ptr as well###--//
xmlDocPtr doc;                  //--Pointer struct for parsing XML--//
xmlChar *pim_xpath_ptr =
    (xmlChar*) "/rpc-reply/pim-join-information/join-family
                /join-group[upstream-state-flags/local-source]
                /multicast-group-address";
xmlChar *temp_xpath_ptr =
    (xmlChar*) "/rpc-reply/igmp-group-information
                /mgm-interface-groups/mgm-group
                [../interface-name = '%s']/multicast-group-address";
xmlChar *igmp_xpath_ptr = NULL;
xmlNodeSetPtr nodeset;
xmlXPathObjectPtr pim_result;   //--Pointer for pim result xml parsing--//
xmlXPathObjectPtr igmp_result;  //--Pointer for igmp result xml parsing--//
xmlChar *keyword_ptr = NULL;    //--Pointer for node text--//
char pim_result_buff[128][64];  //--Char array to store pim xPath results--//
char igmp_result_buff[128][64]; //--Char array to store igmp xPath results--//

//--js commands--//
char js_handshake1[64] = "<?xml version='1.0' encoding='us-ascii'>\n";
char js_handshake2[128] = "<junoscript version='1.0'
    hostname='client1' release='8.4R1'>\n";
char js_login[512] = "<rpc>\n<request-login>\n<username>lab</username>
    \n<challenge-response>Lablab</challenge-response>
    \n</request-login>\n</rpc>\n";
char js_show_pim[512] = "<rpc>\n<get-pim-join-information>
    \n<extensive/></get-pim-join-information></rpc>\n";
char js_show_igmp[512] = "<rpc>\n<get-igmp-group-information/>\n</rpc>\n";
char js_rmv_group[512] = "<rpc>\n<load-configuration>\n<configuration>
    \n<protocols>\n<igmp>\n<interface>\n<name>%s</name>
    \n<static>\n<group delete='delete'>\n<name>%s</name>
    \n</group>\n</static>\n</interface>\n</igmp>\n</protocols>
    \n</configuration>\n</load-configuration>\n</rpc>\n\n\n\n";
char js_add_group[512] = "<rpc>\n<load-configuration>
    \n<configuration>\n<protocols>\n<igmp>
    \n<interface>\n<name>%s</name>\n<static>
    \n<group>\n<name>%s</name>\n<source>
    \n<name>%s</name>\n</source>\n</group>\n</static>
    \n</interface>\n</igmp>\n</protocols>\n</configuration>
    \n</load-configuration>\n</rpc>\n";
char js_commit[64] = "<rpc>\n<commit-configuration>\n</rpc>\n";

//--Function prototypes--//
void error(char *msg);           //--Support error messaging--//
xmlDocPtr getdoc(char *buffer);  //--Parses XML content and loads it into
memory--//
xmlXPathObjectPtr getnodeset (xmlDocPtr doc, xmlChar *xpath);
                                //--Parses xml content for result node(s) from XPath search--//

//--Functions--//
void error(char *msg) {
    perror(msg);
    exit(0);
}

xmlDocPtr getdoc(char *buffer) {

```

```
xmlDocPtr doc;

doc = xmlReadMemory(buffer, strlen((char *)buffer), "temp.xml", NULL, 0);
if (doc == NULL ) {
    fprintf(stderr,"Document not parsed successfully. \n");
    return NULL;
} else {
    #ifdef PRINT
    printf("Document parsed successfully. \n");
    #endif
}
return doc;
}

xmlXPathObjectPtr getnodeset (xmlDocPtr doc, xmlChar *xpath) {

    xmlXPathContextPtr context;
    xmlXPathObjectPtr result;

    context = xmlXPathNewContext(doc);
    if (context == NULL) {
        printf("Error in xmlXPathNewContext\n");
        return NULL;
    }
    result = xmlXPathEvalExpression(xpath, context);
    xmlXPathFreeContext(context);
    if (result == NULL) {
        printf("Error in xmlXPathEvalExpression\n");
        return NULL;
    }
    if(xmlXPathNodeSetIsEmpty(result->nodesetval)) {
        xmlXPathFreeObject(result);
        #ifdef PRINT
        printf("No result\n");
        #endif
        return NULL;
    }
    return result;
}

/--Main--//
int main(int argc, char **argv) {

    if(argc != 4) {
        printf("\nUsage: %s <device Address> <Interface Name>
        <Multicast Source>\n\n", argv[0]);
        exit(0);
    } else {
        strcpy(jserver, argv[1]);
        strcpy(minterface, argv[2]);
        strcpy(msource, argv[3]);
    }
    igmp_xpath_ptr = (xmlChar *)realloc((xmlChar *)igmp_xpath_ptr, 1024);
    sprintf(igmp_xpath_ptr, temp_xpath_ptr, minterface);

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    server = gethostbyname(jserver);
    bzero((char*) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char*) server->h_addr, (char*)
        &serv_addr.sin_addr.sin_addr, server->h_length);
```

```

serv_addr.sin_port = htons(jport);

//--Connect to the js server--//
if(connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
{
    printf("Socket connect error\n");
}

if(fcntl(sockfd, F_SETOWN, getpid()) < 0)
error("Unable to set process owner to us\n");
printf("\nConnected to %s on port %d\n", jserver, jport);

//--Read data from the initial connect--//
sock_bytes = read(sockfd, rcvbuffer, 255);
#ifdef PRINT
printf("\n%s", rcvbuffer);
#endif

//--js initialization handshake--//
sock_bytes = write(sockfd, js_handshake1, strlen(js_handshake1));
    //--Send xml PI to js server--//
sock_bytes = write(sockfd, js_handshake2, strlen(js_handshake2));
    //--Send xml version and encoding to js server--//
sock_bytes = read(sockfd, rcvbuffer, 255);
    //--Read return data from sock buffer--//
rcvbuffer[sock_bytes] = 0;
printf("XML connection to the Junos XML protocol server has been
initialized\n");
#ifdef PRINT
printf("\n%s", rcvbuffer);
#endif

//--js login--//
sock_bytes = write(sockfd, js_login, strlen(js_login));
    //--Send js command--//
while(strstr(small_buff, "superuser") == NULL) {
    //--Continue to read from the buffer until match--//
    sock_bytes = read(sockfd, rcvbuffer, 255);
    rcvbuffer[sock_bytes] = 0;
    strcat(small_buff, rcvbuffer);
    //--Copy buffer contents into pim_buffer--//
}
printf("Login completed to the Junos XML protocol server\n");
#ifdef PRINT
printf("%s\n", small_buff); //--Print the small buff contents--//
#endif
//regfree(&regex_struct);
bzero(small_buff, strlen(small_buff));
    //--Erase small buffer contents--//

//--Begin the for loop here--//
printf("Running continuous IGMP and PIM group comparison...\n\n");
for(;;) {
    //--Begin infinite for loop--//

    //--Get PIM join information--//
    pim_output_ptr = (char *)realloc((char *)pim_output_ptr,
        strlen(js_handshake1));
        //--Allocate memory for xml PI concatenation --//
        //--to pim_output_ptr--//
    strcpy(pim_output_ptr, js_handshake1);
        //--Copy PI to pim_output_ptr--//

```

```

sock_bytes = write(sockfd, js_show_pim, strlen(js_show_pim));
    //--Send show pim joins command--//
while(strstr(pim_output_ptr, "</rpc-reply>") == NULL) {
    //--Continue to read from the buffer until match--//
    sock_bytes = read(sockfd, rcvbuffer, 255);
    //--Read from buffer--//
    rcvbuffer[sock_bytes] = 0;
    pim_output_len = strlen((char *)pim_output_ptr);
    //--Determine current string length of pim_output_ptr--//
    pim_output_ptr = (char *)realloc((char *)pim_output_ptr,
        strlen(rcvbuffer)+pim_output_len);
    //--Reallocate memory for additional data--//
    strcat(pim_output_ptr, rcvbuffer);
    //--Copy data from rcvbuffer to pim_output_ptr--//
}

//--Remove the xmlns entry--//
xmlns_start_ptr = strstr(pim_output_ptr, "xmlns=\"http:");
    //--Find the start of the xmlns entry--pointer --//
    //--returned by strstr()--//
xmlns_end_ptr = strstr(xmlns_start_ptr, ">");
    //--Find the end of the xmlns entry--pointer --//
    //--returned by strstr()--//
repl_str_len = xmlns_end_ptr - xmlns_start_ptr;
    //--Determine the length of the string to be replaced--//
orig_len = strlen((char *)pim_output_ptr) + 1;
    //--Determine the original length of pim_output--//
up_to_len = xmlns_start_ptr - pim_output_ptr;
    //--Determine the length up to the beginning --//
    //--of the xmlns entry--//
remain_len = orig_len - (up_to_len + repl_str_len);
    //--Determine what the remaining length is minus --//
    //--what we are removing--//
memcpy(xmlns_start_ptr - 1, xmlns_start_ptr + repl_str_len, remain_len);

    //--copy the remaining string to the beginning --//
    //--of the replacement string--//
#ifdef PRINT
printf("\n%s\n", pim_output_ptr);
#endif
//--End of GET PIM join information--//

//--Get IGMP membership information--//
igmp_output_ptr = (char *)realloc((char *)igmp_output_ptr,
    strlen(js_handshake1));
strcpy(igmp_output_ptr, js_handshake1);
sock_bytes = write(sockfd, js_show_igmp, strlen(js_show_igmp));
while(strstr(igmp_output_ptr, "</rpc-reply>") == NULL) {
    sock_bytes = read(sockfd, rcvbuffer, 255);
    rcvbuffer[sock_bytes] = 0;
    igmp_output_len = strlen((char *)igmp_output_ptr);
    igmp_output_ptr = (char *)realloc((char *)igmp_output_ptr,
        strlen(rcvbuffer)+igmp_output_len);
    strcat(igmp_output_ptr, rcvbuffer);
}
#ifdef PRINT
printf("\n%s\n", igmp_output_ptr);
#endif
//--End of GET IGMP membership information--//

//--Store XPath results for pim buffer search--//

```

```

doc = getdoc(pim_output_ptr);
    //--Call getdoc() to parse XML in pim_output--//
pim_result = getnodeset (doc, pim_xpath_ptr);
    //--Call getnodeset() which provides xPath result--//
if (pim_result) {
    nodeset = pim_result->nodesetval;
    for (count_a=0; count_a < nodeset->nodeNr; count_a++) {
        //--Run through all node values found--//
        keyword_ptr = xmlNodeListGetString
            (doc, nodeset->nodeTab[count_a]->xmlChildrenNode, 1);
        strcpy(pim_result_buff[count_a], (char *)keyword_ptr);
        //--Copy each node value to its own array element--//
#ifdef PRINT
        printf("PIM Groups: %s\n", pim_result_buff[count_a]);
        //--Print the node value--//
#endif

        xmlFree(keyword_ptr);    //--Free memory used by keyword_ptr--//
        xmlChar *keyword_ptr = NULL;
    }
    xmlXPathFreeObject(pim_result);
    //--Free memory used by result--//
}
xmlFreeDoc(doc);                //--Free memory used by doc--//
xmlCleanupParser();              //--Clean everything else--//
//--End of xPath search--//

//--Store xPath results for igmp buffer search--//
doc = getdoc(igmp_output_ptr);
igmp_result = getnodeset (doc, igmp_xpath_ptr);
if (igmp_result) {
    nodeset = igmp_result->nodesetval;
    for (count_a=0; count_a < nodeset->nodeNr; count_a++) {
        keyword_ptr = xmlNodeListGetString
            (doc, nodeset->nodeTab[count_a]->xmlChildrenNode, 1);
        strcpy(igmp_result_buff[count_a], (char *)keyword_ptr);
#ifdef PRINT
        printf("IGMP Groups: %s\n", igmp_result_buff[count_a]);
#endif

        xmlFree(keyword_ptr);
        xmlChar *keyword_ptr = NULL;
    }
    xmlXPathFreeObject(igmp_result);
}
xmlFreeDoc(doc);
xmlCleanupParser();
//--End of xPath search--//

//--Code to compare pim groups to configured igmp static membership--//
conf_chg = 0;
count_x=0;                        //--Track pim groups--//
count_y=0;                        //--Track igmp groups--//
count_z=0;                        //--Track matches (if set to 1, igmp group matched
pim group)--//
while(strstr(pim_result_buff[count_x], "2") != NULL) {
    //--Run through igmp pim groups--//
    if(strstr(igmp_result_buff[count_y], "2") == NULL) {
        count_z = 0;
        conf_chg = 1;
    }
}

```

```

while(strstr(igmp_result_buff[count_y], "2") != NULL) {
    //--For each pim group, run through all igmp groups--//
    if(strcmp(igmp_result_buff[count_y], pim_result_buff[count_x]) == 0)
    {
        //--If igmp group matches pim group, set z to 1 --//
        //-- (ie count_z=1; --//
        //--Set z to 1 if there was a match (ie - the static --//
        //--membership is configured)--//
    }
    count_y++;          //--Increment igmp result buffer--//
}
if(count_z == 0) {      //--If no igmp group matched the --//
                        //--pim group (z stayed at 0), configure--//

                        //--static membership--//
    printf("Adding this group to igmp: %s\n", pim_result_buff[count_x]);
    sprintf(temp_buff, js_add_group, minterface,
            pim_result_buff[count_x], msource);
    //--Copy js_add_group with pim group to temp_buff--//
#ifdef PRINT
    printf("%s", temp_buff);
#endif
    sock_bytes = write(sockfd, temp_buff, strlen(temp_buff));

    while(strstr(small_buff, "</rpc-reply>") == NULL) {
        sock_bytes = read(sockfd, rcvbuffer, 255);
        rcvbuffer[sock_bytes] = 0;
        strcat(small_buff, rcvbuffer);
    }
#ifdef PRINT
    printf("%s\n", small_buff);
#endif
    bzero(small_buff, strlen(small_buff));
    //--Erase (copy all 0's) small buffer contents--//
    bzero(temp_buff, strlen(temp_buff));
    //--Erase temp_buff contents--//
    conf_chg = 1;
    //--Set conf_chg value to 1 to signify that a --//
    //--commit is needed--//
}
count_x++;              //--increment pim result buffer--//
count_y=0;              //--reset igmp result buffer to start--//
                        //-- at first element--//
count_z=0;              //--reset group match to 0 --//
                        //--(config needed due to no match)--//
}

//--Code for comparing igmp static membership to pim groups--//
count_x=0;
count_y=0;
count_z=0;
while(strstr(igmp_result_buff[count_y], "2") != NULL) {
    if(strstr(pim_result_buff[count_x], "2") == NULL) {
        count_z = 0;
        conf_chg = 1;
    }
    while(strstr(pim_result_buff[count_x], "2") != NULL) {
        if(strcmp(pim_result_buff[count_x], igmp_result_buff[count_y]) == 0)
        {
            count_z = 1;
        }
    }
}

```



```

        count_x++;
    }
    if(count_z == 0) {
        printf("Removing this group from igmp: %s\n",
            igmp_result_buff[count_y]);
        sprintf(temp_buff, js_rmv_group, minterface, igmp_result_buff[count_y]);

        #ifdef PRINT
        printf("%s", temp_buff);
        #endif
        sock_bytes = write(sockfd, temp_buff, strlen(temp_buff));

        while(strstr(small_buff, "</rpc-reply>") == NULL) {
            sock_bytes = read(sockfd, rcvbuffer, 255);
            rcvbuffer[sock_bytes] = 0;

            strcat(small_buff, rcvbuffer);
        }
        #ifdef PRINT
        printf("%s\n", rcvbuffer);
        #endif
        bzero(small_buff, strlen(small_buff));
        bzero(temp_buff, strlen(temp_buff));
        conf_chg = 1;
    }
    count_y++;
    count_x=0;
    count_z=0;
}

if(conf_chg == 1) {
    sock_bytes = write(sockfd, js_commit, strlen(js_commit));
    while(strstr(small_buff, "</rpc-reply>") == NULL) {
        sock_bytes = read(sockfd, rcvbuffer, 255);
        rcvbuffer[sock_bytes] = 0;

        strcat(small_buff, rcvbuffer);
    }
    bzero(small_buff, strlen(small_buff));
    printf("\nCommitted configuration change\n");
} else {
    #ifdef PRINT
    printf("\nNo configuration changes made\n");
    #endif
}
#ifdef PRINT
printf("\n%s\n", small_buff);
#endif

//--Cleanup before next round of checks--//
bzero(rcvbuffer, strlen(rcvbuffer));
//--Erase contents of rcvbuffer--//
char *xmlns_start_ptr = NULL;
//--Nullify the contents--//
char *xmlns_end_ptr = NULL;
//--Nullify the contents--//
for(count_x = 0; count_x < 129; count_x++) {
    //--Erase contents of both pim_result_buff and
    igmp_result_buff--//
    bzero(pim_result_buff[count_x], strlen(pim_result_buff[count_x]));
}

```

```
        bzero(igmp_result_buff[count_x], strlen(igmp_result_buff[count_x]));  
    }  
}
```

- Related Documentation**
- [Establishing a Junos XML Protocol Session Using C Client Applications on page 369](#)
 - [Writing Junos XML Protocol Perl Client Applications on page 342](#)

PART 6

Configuration Statements and Operational Commands

- [Configuration Statements on page 381](#)
- [Operational Commands on page 387](#)

CHAPTER 19

Configuration Statements

- [ephemeral](#) on page 382
- [instance \(Ephemeral Database\)](#) on page 384
- [traceoptions \(NETCONF and Junos XML Protocol\)](#) on page 385

ephemeral

Syntax ephemeral {
 instance *instance-name*;
 allow-commit-synchronize-with-gres;
 ignore-ephemeral-default;
 }

Hierarchy Level [edit system configuration-database]

Release Information Statement introduced in Junos OS Release 16.2R2 on MX Series and T Series routers.
Statement introduced in Junos OS Release 17.3R1 on SRX300, SRX320, SRX340, SRX345, SRX550M, SRX1500, SRX4100, SRX4200, SRX5400, SRX5600, and SRX5800 devices and vSRX instances.
Statement introduced in Junos OS Release 18.1R1 on EX2300, EX3400, EX4300, EX4600, and EX9200 switches.

Description Configure settings for the ephemeral configuration database.

The ephemeral database is an alternate configuration database that enables Juniper Extension Toolkit (JET) applications and NETCONF and Junos XML protocol client applications to simultaneously load and commit configuration changes on devices running Junos OS and with significantly greater throughput than when committing data to the candidate configuration database. Devices running Junos OS provide a default ephemeral database instance as well as the ability to configure up to eight user-defined instances of the ephemeral configuration database.

The ephemeral database is not subject to the same verification required in the static configuration database. As a result, the ephemeral configuration database does not support configuration groups or interface ranges, or macros, commit scripts, or translation scripts. Additionally, certain configuration statements cannot be configured through the ephemeral database as described in [“Unsupported Configuration Statements in the Ephemeral Configuration Database” on page 242](#). Junos OS validates the syntax but does not validate the semantics of configuration data committed to the ephemeral database. Therefore, all configuration data must be validated before loading it into the ephemeral database and committing it on the device.



NOTE: When you configure statements at the [edit system configuration-database ephemeral] hierarchy level and commit the configuration, all Junos OS processes must check and evaluate their complete configuration, which might cause a spike in CPU utilization, potentially impacting other critical software processes.



NOTE: If the ephemeral configuration database is present on a device running Junos OS, commit operations on the static configuration database might

take longer, because additional operations must be performed to merge the static and ephemeral configuration data.

We do *not* recommend using the ephemeral database on devices that have graceful Routing Engine switchover (GRES) enabled. If you elect to use the ephemeral database when GRES is enabled, you must explicitly configure the **allow-commit-synchronize-with-gres** statement to enable the device to synchronize ephemeral configuration data during a commit synchronize operation. If GRES is enabled, and you do not configure the **allow-commit-synchronize-with-gres** statement, the device does not synchronize ephemeral configuration data under any circumstance.

Ephemeral configuration data does not persist across reboots. In addition, when you install a package that requires rebuilding the Junos OS schema, such as an OpenConfig or YANG package, the device deletes all ephemeral configuration data in the process of rebuilding the schema.

Options **allow-commit-synchronize-with-gres**—Enable a device to synchronize ephemeral configuration data to the other Routing Engine when GRES is enabled on the device and a commit synchronize operation is requested.

ignore-ephemeral-default—Disable the default instance of the ephemeral configuration database.


The remaining statements are explained separately. See [CLI Explorer](#).

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


Related Documentation

- [Understanding the Ephemeral Configuration Database on page 237](#)
- [Enabling and Configuring Instances of the Ephemeral Configuration Database on page 244](#)
- [Committing an Instance of the Ephemeral Configuration Database Using the NETCONF or Junos XML Protocol on page 249](#)

instance (Ephemeral Database)

Syntax	<code>instance <i>instance-name</i>;</code>
Hierarchy Level	[edit system configuration-database ephemeral]
Release Information	<p>Statement introduced in Junos OS Release 16.2R2 on MX Series and T Series routers.</p> <p>Statement introduced in Junos OS Release 17.3R1 on SRX300, SRX320, SRX340, SRX345, SRX550M, SRX1500, SRX4100, SRX4200, SRX5400, SRX5600, and SRX5800 devices and vSRX instances.</p> <p>Statement introduced in Junos OS Release 18.1R1 on EX2300, EX3400, EX4300, EX4600, and EX9200 switches.</p>
Description	<p>Enable an instance of the ephemeral configuration database.</p> <p>You can configure up to eight user-defined instances of the ephemeral configuration database. The order in which the instances are listed in the configuration determines the order of their priority when merging conflicting configuration statements from different instances into the configuration. The instances are listed in order from highest to lowest priority. In addition, user-defined instances of the ephemeral configuration database have higher priority than the default ephemeral database instance, which has higher priority than the static configuration database.</p>
	<div> NOTE: When you configure an ephemeral instance, you can specify its placement in the configuration by using the <code>insert</code> command instead of the <code>set</code> command.</div>
Options	<i>instance-name</i> —User-defined name for an instance of the ephemeral configuration database.
Required Privilege Level	<p>maintenance—To view this statement in the configuration.</p> <p>maintenance—To add this statement to the configuration.</p>
Related Documentation	<ul style="list-style-type: none">• Enabling and Configuring Instances of the Ephemeral Configuration Database on page 244• <i>Example: Configuring the Ephemeral Configuration Database Using NETCONF</i>• Understanding the Ephemeral Configuration Database on page 237

traceoptions (NETCONF and Junos XML Protocol)

Syntax	<pre> traceoptions { file <filename> <files number> <match regular-expression> <size size> <world-readable no-world-readable>; flag flag; no-remote-trace; on-demand; } </pre>
Hierarchy Level	[edit system services netconf]
Release Information	<p>Statement introduced in Junos OS Release 12.2.</p> <p>Support for Junos XML protocol sessions added in Junos OS Release 16.1.</p> <p>Option flag debug introduced in Junos OS Release 16.1.</p>
Description	<p>Define tracing operations for NETCONF and Junos XML protocol sessions.</p> <hr/> <div>  <p>NOTE: Starting in Junos OS Release 16.1, when you enable tracing operations at the [edit system services netconf traceoptions] hierarchy, Junos OS enables tracing operations for both NETCONF and Junos XML protocol sessions and adds the [NETCONF] and [JUNOScript] tags to the log file entries to distinguish the type of session. Prior to Junos OS Release 16.1, only NETCONF session data was logged, and the [NETCONF] tag was omitted.</p> </div> <hr/>
Default	If you do not include this statement, NETCONF and Junos XML protocol-specific tracing operations are not performed.
Options	<p>file filename—Name of the file to receive the output of the tracing operation. All files are placed in the /var/log directory.</p> <p>Default: /var/log/netconf</p> <p>files number—(Optional) Maximum number of trace files. When a trace file named trace-file reaches its maximum size, it is renamed and compressed to trace-file.0.gz. When trace-file again reaches its maximum size, trace-file.0.gz is renamed trace-file.1.gz, and trace-file is renamed and compressed to trace-file.0.gz. This renaming scheme continues until the maximum number of trace files is reached. Then the oldest trace file is overwritten.</p> <p>If you specify a maximum number of files, you also must specify a maximum file size with the size option and a filename.</p> <p>Range: 2 through 1000 files</p> <p>Default: 10 files</p>

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

- **all**—Log all incoming and outgoing data from NETCONF and Junos XML protocol sessions.
- **debug**—Log debug level information. Using the **flag all** option is recommended.
- **debug**—Log debug level information. Using the **flag all** option is recommended.
- **incoming**—Log all incoming data from NETCONF and Junos XML protocol sessions.
- **outgoing**—Log all outgoing data from NETCONF and Junos XML protocol sessions.

match *regular-expression*—(Optional) Refine the output to include only those lines that match the regular expression.

no-remote-trace—(Optional) Disable remote tracing.

no-world-readable—(Optional) Disable unrestricted file access, which restricts file access to the owner. This is the default.

on-demand—(Optional) Enable on-demand tracing, which requires that you start and stop tracing operations from within the NETCONF or Junos XML protocol session. If configured, tracing operations are performed for a session only when requested through the **<request-netconf-trace>** operation.

Within a session, issue the **<request-netconf-trace><start/></request-netconf-trace>** RPC to start tracing operations for that session, and issue the **<request-netconf-trace><stop/></request-netconf-trace>** RPC to stop tracing operations for that session.

size *size*—(Optional) Maximum size of each trace file in bytes, kilobytes (KB), megabytes (MB), or gigabytes (GB). If you don't specify a unit, the default is bytes. If you specify a maximum file size, you also must specify a maximum number of trace files with the **files** option and a filename.

Syntax: ***size*** to specify bytes, ***sizek*** to specify KB, ***sizem*** to specify MB, or ***sizeg*** to specify GB

Range: 10240 through 1073741824 bytes

Default: 128 KB

world-readable—(Optional) Enable unrestricted file access.

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


Related Documentation	<ul style="list-style-type: none">• NETCONF and Junos XML Protocol Tracing Operations Overview on page 95• Example: Tracing NETCONF and Junos XML Protocol Session Operations on page 96• <i>netconf</i>
------------------------------	--

CHAPTER 20

Operational Commands

- `show ephemeral-configuration`

show ephemeral-configuration

Syntax	show ephemeral-configuration <instance-name>
Release Information	<p>Command introduced in Junos OS Release 16.2R2 on MX Series and T Series routers.</p> <p>Command introduced in Junos OS Release 17.3R1 on SRX300, SRX320, SRX340, SRX345, SRX550M, SRX1500, SRX4100, SRX4200, SRX5400, SRX5600, and SRX5800 devices and vSRX instances.</p> <p>Command introduced in Junos OS Release 18.1R1 on EX2300, EX3400, EX4300, EX4600, and EX9200 switches.</p>
Description	Display configuration data committed to a given instance of the ephemeral configuration database.
	<div>  <p>TIP: Use the <code>show ephemeral-configuration display merge</code> command to display the configuration data in all instances of the ephemeral database merged with the complete post-inheritance view of the static configuration database.</p> </div>
Options	<p>none—Display the configuration committed to the default instance of the ephemeral configuration database.</p> <p>instance-name—(Optional) Name of a user-defined ephemeral instance for which to display the committed ephemeral configuration data.</p>
Required Privilege Level	view
Related Documentation	<ul style="list-style-type: none"> • (pipe) • Pipe () Filter Functions in the Junos OS Command-Line Interface • Enabling and Configuring Instances of the Ephemeral Configuration Database on page 244
List of Sample Output	<p>show ephemeral-configuration on page 388</p> <p>show ephemeral-configuration eph1 on page 389</p>

Sample Output

show ephemeral-configuration

```

user@host> show ephemeral-configuration
## Last changed: 2017-02-12 17:15:48 PDT
protocols {
  mp1s {
    label-switched-path to-cust1 {

```

```
        to 198.51.100.1;
    }
}
```

show ephemeral-configuration eph1

```
user@host> show ephemeral-configuration eph1
## Last changed: 2017-02-10 13:20:32 PDT
protocols {
  mpls {
    label-switched-path to-hastings {
      to 192.0.2.1;
    }
  }
}
```

