Junos® OS

NETCONF XML Management Protocol Developer Guide

Modified: 2019-06-07

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

Junos® OS NETCONF XML Management Protocol Developer Guide

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

YEAR 2000 NOTICE

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

END USER LICENSE AGREEMENT

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://support.juniper.net/support/eula/. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

Table of Contents

List of Figures

List of Tables

About the Documentation

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.

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.xsl;
       }
     }
   }
   interfaces {
     fxp0 {
       disable;
       unit 0 {
         family inet {
           address 10.0.0.1/24;
         }
       }
     }
   }
   ```

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

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

Merging a Snippet

To merge a snippet, follow these steps:

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

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

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

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

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

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

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

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

Documentation Conventions

Table 1 on page xix 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`
Italic text like this	• Introduces or emphasizes important new terms. • Identifies guide names. • Identifies RFC and Internet draft titles.	• A policy *term* is a named structure that defines match conditions and actions. • *Junos OS CLI User Guide* • RFC 1997, *BGP Communities Attribute*
Italic text like this	Represents variables (options for which you substitute a value) in commands or configuration statements.	Configure the machine's domain name: [edit] root@# **set system domain-name** *domain-name*
Text like this	Represents names of configuration statements, commands, files, and directories; configuration hierarchy levels; or labels on routing platform components.	• To configure a stub area, include the **stub** statement at the **[edit protocols ospf area area-id]** hierarchy level. • The console port is labeled **CONSOLE**.
< > (angle brackets)	Encloses optional keywords or variables.	**stub <default-metric** *metric***>;**
| (pipe symbol)	Indicates a choice between the mutually exclusive keywords or variables on either side of the symbol. The set of choices is often enclosed in parentheses for clarity.	**broadcast | multicast** **(***string1* **|** *string2* **|** *string3***)**
# (pound sign)	Indicates a comment specified on the same line as the configuration statement to which it applies.	**rsvp { # Required for dynamic MPLS only**
[ ] (square brackets)	Encloses a variable for which you can substitute one or more values.	**community name members [** *community-ids* **]**

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

; (semicolon)	Identifies a leaf statement at a configuration hierarchy level.	

GUI Conventions

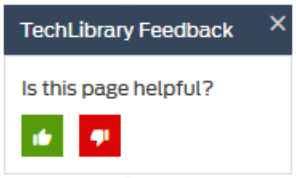
Table 2: Text and Syntax Conventions (continued)

Convention	Description	Examples
Bold text like this	Represents graphical user interface (GUI) items you click or select.	• In the Logical Interfaces box, select **All Interfaces**. • To cancel the configuration, click **Cancel**.
> (bold right angle bracket)	Separates levels in a hierarchy of menu selections.	In the configuration editor hierarchy, select **Protocols>Ospf**.

Documentation Feedback

We encourage you to provide feedback so that we can improve our documentation. You can use either of the following methods:

- Online feedback system—Click TechLibrary Feedback, on the lower right of any page on the Juniper Networks TechLibrary site, and do one of the following:



  - Click the thumbs-up icon if the information on the page was helpful to you.
  - Click the thumbs-down icon if the information on the page was not helpful to you or if you have suggestions for improvement, and use the pop-up form to provide 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 Juniper Care or Partner Support Services 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/
- Create a service request online: https://myjuniper.juniper.net

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

Creating a Service Request with JTAC

You can create a service request with JTAC on the Web or by telephone.

- Visit https://myjuniper.juniper.net.
- 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://support.juniper.net/support/requesting-support/.

PART 1

Overview

CHAPTER 1

NETCONF XML Management Protocol Overview

NETCONF XML Management Protocol and Junos XML API Overview

The NETCONF XML management protocol is an XML-based protocol that client applications use to request and change configuration information on routing, switching, and security devices. It uses an Extensible Markup Language (XML)-based data encoding for the configuration data and remote procedure calls. The NETCONF protocol defines basic operations that are equivalent to configuration mode commands in the 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 to perform those operations.

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

The NETCONF XML management protocol is described in RFC 6241, *Network Configuration Protocol (NETCONF)*, which is available at https://tools.ietf.org/html/rfc6241.

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

request tag. The NETCONF 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 NETCONF XML management protocol and Junos XML API to configure devices running Junos OS or to request information about the device configuration or operation. You can write client applications to interact with the NETCONF server, and you can also use the NETCONF XML management 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 NETCONF 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 NETCONF XML Management Protocol and Junos XML API

The NETCONF XML management protocol and Junos XML API fully document all options for every supported Junos OS operational request and all elements in every Junos OS configuration statement. The tag names clearly indicate the function of an element in an operational request or 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. NETCONF and Junos XML 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. It compares formatted ASCII and XML-tagged versions of output from a device running the 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, which is

```
, SNMP ifIndex
```

A problem arises if the format or ordering of output changes in a later version of the Junos OS, for example, if a **Logical index** field is added following the interface index number:

```
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 following label instead:

```
, Logical index
```

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 NETCONF 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. 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 design and write special routines and data structures in your display program to extract and store the information needed 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

- NETCONF XML Management Protocol and Junos XML API Overview on page 3
- XML Overview on page 9

CHAPTER 2

NETCONF and Junos XML Tags Overview

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

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*
- XML and Junos OS Overview on page 7

XML and NETCONF XML Management Protocol Conventions Overview

A client application must comply with XML and NETCONF 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 NETCONF and Junos XML document type definitions (DTD)s for the kind of information encoded in the request. The client application must emit tag elements in the required order and only in the legal contexts. Compliant applications are easier to maintain in the event of changes to the Junos OS or NETCONF protocol.

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

The following sections describe NETCONF XML management protocol conventions:

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

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 NETCONF Server" on page 50.

A *response* tag element represents the NETCONF 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 NETCONF Server Response" on page 52.

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

Client Application

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

NETCONF Server

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

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 NETCONF 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 NETCONF Server Response" on page 52. For information about the **xmlns** attribute in the opening **<interface-information>** tag, see "Requesting Operational Information Using NETCONF" on page 203. For information about the **]]>]]>** character sequence, see "Generating Well-Formed XML Documents" on page 31.

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>**. For more information, see "Mapping for Objects That Have an Identifier" on page 20.

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 NETCONF 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 NETCONF 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 NETCONF 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 NETCONF 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 NETCONF server preserves the white space in the configuration database.

In its responses, the NETCONF 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 NETCONF 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 NETCONF server gracefully but must not depend on their content. Client applications also cannot use comments to convey information to the NETCONF 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. -->
```

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 NETCONF 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)	&amp;
> (greater-than sign)	&gt;
< (less-than sign)	&lt;

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)	&amp;
' (apostrophe)	&apos;
> (greater-than sign)	&gt;
< (less-than sign)	&lt;
" (quotation mark)	&quot;

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

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

The **<condition>** tag 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>** tag 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 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	<rpc> <get-interface-information> <interface-name>t3-5/1/0:0</interface-name> </get-interface-information> </rpc>
show bgp neighbor 10.168.1.122	<rpc> <get-bgp-neighbor-information> <neighbor-address>10.168.1.122</neighbor-address> </get-bgp-neighbor-information> </rpc>

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	<rpc> <get-isis-adjacency-information> <detail/> </get-isis-adjacency-information> </rpc>

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*

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

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

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

JUNOS XML Tags

```
<configuration>
   <system>
      <login>
         <!- - tags for child statements - ->
      </login>
   </system>
   <protocols>
      <ospf>
         <!- - 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

```
protocols {
  bgp {
    group G1 {

      type external;
      peer-as 56;
      neighbor 10.0.0.1;


    }
    group G2 {

      type external;
      peer-as 57;
      neighbor 10.0.10.1;


    }
  }
}
```

JUNOS XML Tags

```
<configuration>
  <protocols>
    <bgp>
      <group>
        <name>G1</name>
        <type>external</type>
        <peer-as>56</peer-as>
        <neighbor>
          <name>10.0.0.1</name>
        </neighbor>
      </group>
      <group>
        <name>G2</name>
        <type>external</type>
        <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 10.0.01 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

```
protocols {
   ospf {

      /* New backbone area */
      area 0.0.0.0 {

         interface so-0/0/0 { # From jnpr1 to jnpr2

            hello-interval 5;
         }
      }
   }
}
```

JUNOS XML Tags

```
<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>
```

T1510

Using NETCONF Configuration Response Tag Elements in NETCONF Requests and Configuration Changes

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

The Junos XML tag elements within the **<configuration>** tag 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 NETCONF 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 NETCONF server's response in 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 NETCONF server's response, it is certain to be syntactically correct.

Similarly, when a client application requests information about a configuration element (hierarchy level or configuration object), it uses the same tag elements that the NETCONF server will return in response. To represent the element, the client application sends a complete stream of tag elements from the top of the configuration hierarchy (represented by the **<configuration>** tag element) down to the requested element. The innermost tag element, which represents the level or object, is either empty or includes the identifier tag element only. The NETCONF 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 NETCONF" on page 213.

The tag streams emitted by the NETCONF server and by a client application can differ in the use of white space, as described in "XML and NETCONF XML Management Protocol Conventions Overview" on page 11.

Related Documentation

- XML and NETCONF XML Management Protocol Conventions Overview on page 11
- Mapping Configuration Statements to Junos XML Tag Elements on page 19
- Requesting Configuration Data Using NETCONF on page 213

PART 2

Managing NETCONF Sessions

CHAPTER 3

NETCONF Session Overview

NETCONF Session Overview

Communication between the NETCONF server and a client application is session based. The server and client explicitly establish a connection and session before exchanging data and close the session and connection when they are finished.

The streams of NETCONF and Junos XML tag elements emitted by the NETCONF server and the client application must each constitute well-formed XML by obeying the structural rules defined in the document type definition (DTD) for the kind of information they are exchanging. The client application must emit tag elements in the required order and only in the allowed contexts.

Client applications access the NETCONF server using the SSH protocol and use the standard SSH authentication mechanism. After authentication, the NETCONF server uses the configured Junos OS login usernames and classes to determine whether a client application is authorized to make each request.

The following list outlines the basic structure of a NETCONF session:

1. The client application establishes a connection to the NETCONF server and opens the NETCONF session.
2. The NETCONF server and client application exchange initialization information, which is used to determine if they are using compatible versions of the Junos OS and the NETCONF XML management protocol.
3. The client application sends one or more requests to the NETCONF server and parses its responses.
4. The client application closes the NETCONF session and the connection to the NETCONF server.

For an example of a complete NETCONF session, see "Sample NETCONF Session" on page 62.

Related Documentation

- Generating Well-Formed XML Documents on page 31
- Connecting to the NETCONF Server on page 45
- Starting the NETCONF Session on page 46
- Sample NETCONF Session on page 62

Understanding the Client Application's Role in a NETCONF Session

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

1. Satisfies the prerequisites for an SSH connection, as described in "Establishing an SSH Connection for a NETCONF Session" on page 35.
2. Establishes a connection to the NETCONF server on the routing, switching, or security platform, as described in "Connecting to the NETCONF Server" on page 45.
3. Opens a NETCONF session, as described in "Starting the NETCONF Session" on page 46.
4. Optionally locks 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. For more information, see "Locking and Unlocking the Candidate Configuration Using NETCONF" on page 58.

   For information about the ephemeral configuration database, see "Understanding the Ephemeral Configuration Database" on page 177 and "Enabling and Configuring Instances of the Ephemeral Configuration Database" on page 184.
5. Requests operational or configuration information, or changes configuration information, as described in "Requesting Operational Information Using NETCONF" on page 203, "Requesting Configuration Data Using NETCONF" on page 213, and "Editing the Configuration Using NETCONF" on page 139.
6. (Optional) Verifies the syntactic correctness of the candidate configuration before attempting to commit it, as described in "Verifying the Candidate Configuration Syntax Using NETCONF" on page 173.
7. Commits changes made to the candidate configuration, as described in "Committing the Candidate Configuration Using NETCONF" on page 174 and "Committing the Candidate Configuration Only After Confirmation Using NETCONF" on page 175, 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 191.
8. Unlocks the candidate configuration if it is locked or closes 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 Using NETCONF" on page 58.

9. Ends the NETCONF session and closes the connection to the device, as described in "Ending a NETCONF Session and Closing the Connection" on page 62.

Generating Well-Formed XML Documents

Each set of NETCONF and Junos XML tag elements emitted by the NETCONF server and a client application within a **<hello>**, **<rpc>**, or **<rpc-reply>** tag element must constitute a well-formed XML document by obeying the structural rules defined in the document type definition (DTD) for the kind of information being sent. The client application must emit tag elements in the required order and only in the allowed contexts.

The NETCONF server and client applications must also comply with RFC 4742, *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, available at http://www.ietf.org/rfc/rfc4742.txt . In particular, the server and applications must send the character sequence **]]>]]>** after each XML document. Because this sequence is not legal within an XML document, it unambiguously signals the end of a document. In practice, the client application sends the sequence after the closing **</hello>** tag and each closing **</rpc>** tag, and the NETCONF server sends it after the closing **</hello>** tag and each closing **</rpc-reply>** tag.

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.

```
<!-- generated by a client application -->
<hello | rpc>
    <!-- contents of top-level tag element  -->
</hello | /rpc>
]]>]]>

<!-- generated by the NETCONF server -->
<hello | rpc-reply attributes>
    <!-- contents of top-level tag element  -->
</hello | /rpc-reply>
]]>]]>
```

Related Documentation

- Connecting to the NETCONF Server on page 45
- Starting the NETCONF Session on page 46

Understanding the Request Procedure in a NETCONF Session

You can use the NETCONF 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 element for many commands in the CLI **show** family of commands.

The tag element for configuration requests is the NETCONF **<get-config>** tag element. 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 NETCONF 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 NETCONF server, a client application performs the procedures described in the indicated sections:

1. Establishes a connection to the NETCONF server on the routing, switching, or security platform, as described in "Connecting to the NETCONF Server" on page 45.
2. Opens a NETCONF session, as described in "Starting the NETCONF Session" on page 46.
3. Optionally locks 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. For more information, see "Locking and Unlocking the Candidate Configuration Using NETCONF" on page 58.

   For information about the ephemeral configuration database, see "Understanding the Ephemeral Configuration Database" on page 177 and "Enabling and Configuring Instances of the Ephemeral Configuration Database" on page 184.
4. Makes any number of requests one at a time, freely intermingling operational and configuration requests. See "Requesting Operational Information Using NETCONF" on page 203 and "Requesting Configuration Data Using NETCONF" on page 213. The application can also intermix requests with configuration changes.
5. Accepts the tag stream emitted by the NETCONF server in response to each request and extracts its content, as described in "Parsing the NETCONF Server Response" on page 52.
6. Unlocks the candidate configuration, if it is locked, or closes 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 Using NETCONF" on page 58

7. Ends the NETCONF session and closes the connection to the device, as described in "Ending a NETCONF Session and Closing the Connection" on page 62.

CHAPTER 4

Managing NETCONF Sessions

Establishing an SSH Connection for a NETCONF Session

Establishing an SSH Connection for a NETCONF Session

You use the SSH protocol to establish connections between a *configuration management server* and a device running Junos OS. A configuration management server, as the name implies, is used to configure the device running Junos OS remotely.

There are two options available when establishing a connection between the configuration management server and a device running Junos OS: SSH and outbound SSH. With SSH, the configuration management server initiates an SSH session with the device running Junos OS. Outbound SSH is used when the configuration management server cannot initiate an SSH connection because of network restrictions (such as a firewall). In this

situation, the device running Junos OS is configured to initiate, establish, and maintain an SSH connection with a predefined set of configuration management servers. For a complete discussion of outbound SSH, see *Configuring Outbound SSH Service* .

Prerequisites for Establishing an SSH Connection for NETCONF Sessions

Before the configuration management server establishes an SSH connection with a device running Junos OS, you must satisfy the requirements discussed in the following sections.

1. Installing SSH Software on the Configuration Management Server on page 36
2. Configuring a User Account for the Client Application on Devices Running Junos OS on page 36
3. Configuring a Public/Private Key Pair or Password for the Junos OS User Account on page 37
4. Accessing the Keys or Password with the Client Application on page 38
5. Enabling NETCONF Service over SSH on page 39

Installing SSH Software on the Configuration Management Server

The configuration management server handles the SSH connection between the configuration management server and the device running Junos OS. Therefore, the SSH software must be installed locally on the configuration management server. For information about obtaining and installing SSH software, see http://www.ssh.com/ and http://www.openssh.com/.

Configuring a User Account for the Client Application on Devices Running Junos OS

When establishing a NETCONF session, the configuration management server must log in to the device running Junos OS. Thus, each configuration management server needs a user account on each device where a NETCONF session will be established. The following instructions explain how to create a login account on devices running Junos OS. Alternatively, you can skip this section and enable authentication through RADIUS or TACACS+.

To determine whether a login account exists on a device running Junos OS, enter 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 to create one:

1. Configure the **user** statement at the **[edit system login]** hierarchy level and specify a username. Include the **class** statement, and specify a login class that has the permissions required for all actions to be performed by the application.

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

2. Optionally, include the **full-name** and **uid** statements at the **[edit system login user *username*]** hierarchy level.

3. Commit the configuration to activate the user account on the device.

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

4. Repeat the preceding steps on each device running Junos OS where the client application establishes NETCONF sessions.

See Also

- *Junos OS User Accounts Overview*

Configuring a Public/Private Key Pair or Password for the Junos OS User Account

The configuration management server needs an SSH public/private key pair, a text-based password, or both before it can authenticate with the NETCONF server. A public/private key pair is sufficient if the account is used only to connect to the NETCONF server through SSH. If the account is also used to access the device in other ways (for login on the console, for example), it must have a text-based password. The password is also used (the SSH server prompts for it) if key-based authentication is configured but fails.

NOTE: You can skip this section if you have chosen to enable authentication through RADIUS or TACACS+.

To create a text-based password, perform the following steps:

1. Include either the **plain-text-password** or **encrypted-password** statement at the **[edit system login user *username* authentication]** hierarchy level.

   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 username 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 username authentication]
user@host# set encrypted-password "password"
```

2. Commit the configuration.

```
[edit system login user username authentication]
```

```
user@host# commit
```

3. Repeat the preceding steps on each device running Junos OS where the client application establishes NETCONF sessions.

To create an SSH public/private key pair, perform the following steps:

1. Issue the **ssh-keygen** command in the standard command shell (not the Junos OS CLI) on the configuration management server where the client application runs.

   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 manual page for the **ssh-keygen** command. Junos OS uses SSH version 2 by default, but also supports version 1.

   ```
   % ssh-keygen options
   ```

2. Associate the public key with the Junos OS login account by including the **load-key-file** statement at the **[edit system login user *account-name* authentication]** hierarchy level.

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

   Junos OS copies the contents of the specified file onto the device running Junos OS. *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 ssh-dsa**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.

3. Commit the configuration.

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

4. Repeat Step 2 and Step 3 on each device running Junos OS where the client application establishes NETCONF sessions.

Accessing the Keys or Password with the Client Application

The client application must be able to access the public/private keys or password you created in "Configuring a Public/Private Key Pair or Password for the Junos OS User Account" on page 37 and provide it when the NETCONF server prompts for it.

There are several methods for enabling the application to access the key or password:

- If public/private keys are used, the ssh-agent program runs on the computer where the client application runs, and handles the private key.
- When a user starts the application, the application prompts the user for the password and stores it temporarily in a secure manner.
- The password is stored in encrypted form in a secure local-disk location or in a secured database.

Enabling NETCONF Service over SSH

RFC 4742, *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, requires that the NETCONF server, by default, provide the client device with access to the NETCONF SSH subsystem when the SSH session is established over a dedicated IANA-assigned TCP port. Use of a dedicated port makes it easy to identify and filter NETCONF traffic. The IANA-assigned port for NETCONF-over-SSH sessions is 830.

You also can configure the server to allow access to the NETCONF SSH subsystem either over the default SSH port (22) or over a port number that is explicitly configured. An explicitly configured port accepts only NETCONF-over-SSH sessions and rejects regular SSH session requests. If SSH services are enabled on the server, the default SSH port (22) continues to accept NETCONF sessions even when an alternate NETCONF-over-SSH port is configured. For added security, you can configure event policies that utilize **UI_LOGIN_EVENT** information to effectively disable the default port or further restrict NETCONF server access on a port.

To enable NETCONF service over SSH on a device running Junos OS, perform the following steps:

1. Include one of the following statements at the indicated configuration hierarchy level:
   - To enable access to the NETCONF SSH subsystem using the default NETCONF-over-SSH port (830) as specified by RFC 4742, include the **netconf ssh** statement at the **[edit system services]** hierarchy level:

     ```
     [edit system services]
     user@host# set netconf ssh
     ```

   - To enable access to the NETCONF SSH subsystem using a specified port number, configure the **port** statement with the desired port number at the [**edit system services netconf ssh**] hierarchy level.

     ```
     [edit system services]
     user@host# set netconf ssh port port-number
     ```

     The ***port-number*** can range from 1 through 65535. The configured port accepts only NETCONF-over-SSH sessions and rejects regular SSH session requests.

NOTE: Although NETCONF-over-SSH can be configured on any port from 1 through 65535, you should avoid configuring access on a port that is normally assigned for another service. This practice avoids potential resource conflicts. If you configure NETCONF-over-SSH on a port assigned for another service, such as FTP, and that service is enabled, a **commit check** does not reveal a resource conflict or issue any warning message to that effect.

- To enable access to the NETCONF SSH subsystem using the default SSH port (22), include the **ssh** statement at the **[edit system services]** hierarchy level. This configuration enables SSH access to the device for all users and applications. The **ssh** statement can be included in the configuration in addition to the configuration statements listed previously.

```
[edit system services]
user@host# set ssh
```

2. Commit the configuration:

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

3. Repeat the preceding steps on each device running Junos OS where the client application establishes NETCONF sessions.

See Also
- *Junos OS User Accounts Overview*

Prerequisites for Establishing an Outbound SSH Connection for NETCONF Sessions

To enable a configuration management server to establish an outbound SSH connection to the NETCONF server, you must satisfy the requirements discussed in the following sections:

1. Configuring the Device Running Junos OS for Outbound SSH on page 40
2. Installing SSH Software on the Client on page 42
3. Receiving and Managing the Outbound SSH Initiation Sequence on the Client on page 43
4. Enabling NETCONF Service over SSH on page 43

Configuring the Device Running Junos OS for Outbound SSH

To configure the device running Junos OS for outbound SSH:

1. At the **[edit system services ssh]** hierarchy level, set the SSH **protocol-version** to v2:

```
[edit system services ssh]
user@host# set protocol-version v2
```

2. Generate or 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.

3. If you are manually installing the public key on the configuration management server, transfer the public key to the configuration management server.

4. At the **[edit system services]** hierarchy level, include the **outbound-ssh** configuration hierarchy and any required statements.

```
[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 per attempt.

NOTE: Starting in Junos OS Release 15.1, Junos OS supports outbound SSH connections with devices that have IPv6 addresses.

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. To configure the keepalive message, you must configure both the **timeout** and **retry** statements.

- **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 (sticky | in-order)—(Optional) Specify the method the device running Junos OS 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.

secret *password*—(Optional) Public SSH host key of the device. 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 devices public key.

services netconf—(Required) Specifies the services available for the session. Currently, NETCONF is the only service available.

5. Commit the configuration:

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

Installing SSH Software on the Client

Once the device establishes the SSH connection to the configuration management server, the configuration management server takes control of the SSH session. Therefore, the SSH client software must be installed locally on the configuration management server. For information about obtaining and installing SSH software, see http://www.ssh.com/ and http://www.openssh.com/ .

Receiving and Managing the Outbound SSH Initiation Sequence on the Client

When configured for outbound SSH, the device running Junos OS attempts to maintain a constant connection with a configuration management server. Whenever an outbound SSH session is not established, the device sends an outbound SSH initiation sequence to a configuration management server listed in the device's configuration management server list. Prior to establishing a connection with the device, each configuration management server must be set up to receive this initiation sequence, establish a TCP connection with the device, and transmit the device identity back to the device.

The initiation sequence takes one of two forms, depending on how you chose to handle the Junos OS server's public key.

If the public key is installed manually on the configuration management server, the initiation sequence takes the following form:

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

If the public key is forwarded to the configuration management server by the device during the initialization sequence, the sequence takes the following form:

```
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
```

Enabling NETCONF Service over SSH

RFC 4742, *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, requires that the NETCONF server, by default, provide the client device with access to the NETCONF SSH subsystem when the SSH session is established over a dedicated IANA-assigned TCP port. Use of a dedicated port makes it easy to identify and filter NETCONF traffic. The IANA-assigned port for NETCONF-over-SSH sessions is 830.

You also can configure the server to allow access to the NETCONF SSH subsystem either over the default SSH port (22) or over a port number that is explicitly configured. An explicitly configured port accepts only NETCONF-over-SSH sessions and rejects regular SSH session requests. If SSH services are enabled on the server, the default SSH port (22) continues to accept NETCONF sessions even when an alternate NETCONF-over-SSH port is configured. For added security, you can configure event policies that utilize **UI_LOGIN_EVENT** information to effectively disable the default port or further restrict NETCONF server access on a port.

To enable NETCONF service over SSH on a device running Junos OS, perform the following steps:

1. Include one of the following statements at the indicated hierarchy level:

- To enable access to the NETCONF SSH subsystem using the default NETCONF-over-SSH port (830) as specified by RFC 4742, include the **netconf ssh** statement at the **[edit system services]** hierarchy level:

  ```
  [edit system services]
  user@host# set netconf ssh
  ```

- To enable access to the NETCONF SSH subsystem using a specified port number, configure the **port** statement with the desired port number at the [**edit system services netconf ssh**] hierarchy level.

  ```
  [edit system services]
  user@host# set netconf ssh port port-number
  ```

  The ***port-number*** can range from 1 through 65535. The configured port accepts only NETCONF-over-SSH sessions and rejects regular SSH session requests.

  NOTE: Although NETCONF-over-SSH can be configured on any port from 1 through 65535, you should avoid configuring access on a port that is normally assigned for another service. This practice avoids potential resource conflicts. If you configure NETCONF-over-SSH on a port assigned for another service, such as FTP, and that service is enabled, a **commit check** does not reveal a resource conflict or issue any warning message to that effect.

- To enable access to the NETCONF SSH subsystem using the default SSH port (22), include the **ssh** statement at the **[edit system services]** hierarchy level. This configuration enables SSH access to the device for all users and applications. The **ssh** statement can be included in the configuration in addition to the configuration statements listed previously.

  ```
  [edit system services]
  user@host# set ssh
  ```

2. Commit the configuration:

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

3. Repeat the preceding steps on each device running Junos OS where the client application establishes a NETCONF session.

See Also

- *Configuring SSH Service for Remote Access to the Router or Switch*
- *Configuring Outbound SSH Service*

Release History Table

Release	Description
15.1	Starting in Junos OS Release 15.1, Junos OS supports outbound SSH connections with devices that have IPv6 addresses.

Related Documentation

- *Junos OS User Accounts Overview*
- *Configuring SSH Service for Remote Access to the Router or Switch*
- *Configuring Outbound SSH Service*

Connecting to the NETCONF Server

Before a client application can connect to the NETCONF server, you must satisfy the requirements described in "Establishing an SSH Connection for a NETCONF Session" on page 35.

When the prerequisites are satisfied, applications written in Perl use the NETCONF Perl module to connect to the NETCONF server. A client application that does not use the NETCONF Perl module uses one of the following methods:

- It uses SSH library routines to establish an SSH connection to the NETCONF server, provide the username and password or passphrase, and create a channel that acts as an SSH subsystem for the NETCONF session. Providing instructions for using library routines is beyond the scope of this document.
- It establishes a NETCONF session using the **ssh** command.
  - To establish a NETCONF session as an SSH subsystem over the default NETCONF port (830), the client application issues the following command:

    ```
    ssh user@hostname -p 830 -s netconf
    ```

    The **-p** option defines the port number on which the NETCONF server listens. This option can be omitted if you enabled access to SSH over the default port in "Enabling NETCONF Service over SSH" on page 39.

    The **-s** option establishes the NETCONF session as an SSH subsystem.
  - To establish a NETCONF session over the default SSH port (22) and use pseudo-tty allocation, the client application issues the following command:

    ```
    ssh user@hostname -t netconf
    ```

    NOTE: Using multiple **-t** options forces pseudo-tty allocation even if SSH has no local tty.

Establishing a NETCONF session as an SSH subsystem with a dedicated port enables a device to more easily identify and filter NETCONF traffic. However, establishing a

NETCONF session over the default SSH port using the **-t** option has the advantage of providing visibility to the session on the device running Junos OS, for example, when issuing the **show system users** operational command.

The application must include code to intercept the NETCONF server's prompt for the password or passphrase. Perhaps the most straightforward method is for the application to use a utility such as the **expect** command. The NETCONF Perl client uses this method, for example.

Related Documentation

- Generating Well-Formed XML Documents on page 31
- Starting the NETCONF Session on page 46

Starting the NETCONF Session

Each NETCONF session begins with a handshake in which the NETCONF server and the client application specify the NETCONF capabilities they support. The following sections describe how to start a NETCONF session:

- Exchanging <hello> Tag Elements on page 46
- Verifying Compatibility on page 48

Exchanging <hello> Tag Elements

The NETCONF server and client application each begin by emitting a **<hello>** tag element to specify which operations, or *capabilities*, they support from among those defined in the NETCONF specification. The **<hello>** tag element encloses the **<capabilities>** tag element and the **<session-id>** tag element, which specifies the UNIX process ID (PID) of the NETCONF server for the session. Within the **<capabilities>** tag element, a **<capability>** element specifies each supported function.

The client application must emit the **<hello>** tag element before any other tag element during the NETCONF session, and must not emit it more than once.

Each capability defined in the NETCONF specification is represented in a **<capability>** tag element by a uniform resource name (URN). Capabilities defined by individual vendors are represented by uniform resource identifiers (URIs), which can be URNs or URLs. The NETCONF XML management protocol emits a **<hello>** tag element similar to the following sample output (each **<capability>** tag element appears on three lines for legibility only):

```
<hello>
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    <capability>
       urn:ietf:params:netconf:capability:confirmed-commit:1.0
    </capability>
    <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
    <capability>
       urn:ietf:params:netconf:capability:url:1.0?scheme=http,ftp,file
    </capability>
```

```
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
    </capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
    </capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:validate:1.0
    </capability>
    <capability>
       urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
    </capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>22062</session-id>
</hello>
```

(For information about the **]]>]]>** character sequence, see "Generating Well-Formed XML Documents" on page 31.)

The URIs in the **<hello>** tag element indicate the following supported capabilities:

- **urn:ietf:params:netconf:base:1.0**—The NETCONF server supports the basic operations and tag elements defined in the base NETCONF specification.
- **urn:ietf:params:netconf:capability:candidate:1.0**—The NETCONF server supports operations on a candidate configuration. For more information, see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214, "Locking and Unlocking the Candidate Configuration Using NETCONF" on page 58, "Editing the Configuration Using NETCONF" on page 139, "Rolling Back Uncommitted Changes in the Candidate Configuration Using NETCONF" on page 155, "Verifying the Candidate Configuration Syntax Using NETCONF" on page 173, and "Committing the Candidate Configuration Using NETCONF" on page 174.
- **urn:ietf:params:netconf:capability:confirmed-commit:1.0**—The NETCONF server supports confirmed commit operations. For more information, see "Committing the Candidate Configuration Only After Confirmation Using NETCONF" on page 175.
- **urn:ietf:params:netconf:capability:validate:1.0**—The NETCONF server supports the validation operation, which verifies the syntactic correctness of a configuration without actually committing it. For more information, see "Verifying the Candidate Configuration Syntax Using NETCONF" on page 173.
- **urn:ietf:params:netconf:capability:url:1.0?protocol=http,ftp,file**—The NETCONF server accepts configuration data stored in a file. It can retrieve files both from its local filesystem (indicated by the **file** option in the URN) and from remote machines by using Hypertext Transfer Protocol (HTTP) or FTP (indicated by the **http** and **ftp** options in the URN). For more information, see "Uploading and Formatting Configuration Data in a NETCONF Session" on page 141.
- **http://xml.juniper.net/netconf/junos/1.0**—The NETCONF server supports the operations defined in the Junos XML API for requesting and changing operational information (the

tag elements in the *Junos XML API Operational Developer Reference*). The NETCONF server also supports operations in the Junos XML management protocol for requesting or changing configuration information.

NETCONF client applications should use only native NETCONF XML management protocol operations and supported extensions available in the Junos XML management protocol for configuration functions. The semantics of corresponding Junos XML protocol operations and NETCONF XML protocol operations are not necessarily identical, so using Junos XML protocol configuration operations other than the documented supported extensions can lead to unexpected results.

- **http://xml.juniper.net/dmi/system/1.0**—The NETCONF server supports the operations defined in the Device Management Interface (DMI) specification.

To comply with the NETCONF specification, the client application also emits a **<hello>** tag element to define the capabilities it supports. It does not include the **<session-id>** tag element:

```
<hello>
<capabilities>
  <capability>first-capability</capability>
  <!-- tag elements for additional capabilities -->
</capabilities>
</hello>
]]>]]>
```

The session continues when the client application sends a request to the NETCONF server. The NETCONF server does not emit any tag elements after session initialization except in response to the client application's requests.

Verifying Compatibility

Exchanging **<hello>** tag elements enables a client application and the NETCONF server to determine if they support the same capabilities. In addition, we recommend that the client application determine the version of the Junos OS running on the NETCONF server. After emitting its **<hello>** tag, the client application emits the **<get-software-information>** tag element in an **<rpc>** tag element:

```
<rpc>
  <get-software-information/>
</rpc>
]]>]]>
```

The NETCONF server returns the **<software-information>** tag element, which encloses the **<host-name>** and **<product-name>** tag elements plus a **<package-information>** tag element for each Junos OS module. The **<comment>** tag element within the **<package-information>** tag element specifies the Junos OS Release number (in the following example, **8.2** for Junos OS Release 8.2) and the build date in the format YYYYMMDD (year, month, day—12 January 2007 in the following example). Some tag elements appear on multiple lines, for legibility only:

```
  <rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" \
```

```
                xmlns:junos="http://xml.juniper.net/junos/8.2R1/junos">
    <software-information>
        <host-name>router1</host-name>
        <product-name>m20</product-name>
        <package-information>
            <name>junos</name>
            <comment>JUNOS Base OS boot [8.2-20070112.0]</comment>
        </package-information>
        <package-information>
            <name>jbase</name>
            <comment>JUNOS Base OS Software Suite \
                                [8.2-20070112.0]</comment>
        </package-information>
        <!-- <package-information> tag elements for additional modules -->
    </software-information>
    </capabilities>
</rpc-reply>
]]>]]>
```

Normally, the version is the same for all Junos OS modules running on the device (we recommend this configuration for predictable routing performance). Therefore, verifying the version number of just one module is usually sufficient.

The client application is responsible for determining how to handle any differences in version or capabilities. For fully automated performance, include code in the client application that determines whether it supports the same capabilities and Junos OS version as the NETCONF server. Decide which of the following options is appropriate when there are differences, and implement the corresponding response:

- Ignore differences in capabilities and Junos OS version, and do not alter the client application's behavior to accommodate the NETCONF server. A difference in Junos OS versions does not necessarily make the server and client incompatible, so this is often a valid approach. Similarly, it is a valid approach if the capabilities that the client application does not support are operations that are always initiated by a client, such as validation of a configuration and confirmed commit. In that case, the client maintains compatibility by not initiating the operation.
- Alter standard behavior to be compatible with the NETCONF server. If the client application is running a later version of the Junos OS, for example, it can choose to emit only NETCONF and Junos XML tag elements that represent the software features available in the NETCONF server's version of the Junos OS.
- End the NETCONF session and terminate the connection. This is appropriate if you decide that it is not practical to accommodate the NETCONF server's version or capabilities. For instructions, see "Ending a NETCONF Session and Closing the Connection" on page 62.

Related Documentation

- Generating Well-Formed XML Documents on page 31
- Connecting to the NETCONF Server on page 45
- Sending Requests to the NETCONF Server on page 50
- Parsing the NETCONF Server Response on page 52

Sending Requests to the NETCONF Server

To initiate a request to the NETCONF server, a client application emits the opening **<rpc>** tag, followed by 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 NETCONF 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 NETCONF 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 NETCONF server echoes the attribute in its opening **<rpc-reply>** tag, making it easy to map the response to the initiating request. The NETCONF specification specifies the name **message-id** for this attribute.

Although operational and configuration requests conceptually belong to separate classes, a NETCONF session does not have distinct modes that correspond to CLI operational and configuration modes. Each request tag element 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 50
- Configuration Information Requests on page 51
- Configuration Change Requests on page 51

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 element for many CLI commands. For example, the **<get-interface-information>** tag element corresponds to the **show interfaces** command, and the **<get-chassis-inventory>** tag element 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 NETCONF" on page 203. 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 switching, routing, or security platform). The candidate and committed configurations diverge when there are uncommitted changes to the candidate configuration.

The NETCONF protocol defines the **<get-config>** 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 from the **[edit system login]** hierarchy level of the candidate configuration:

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter type="subtree">
      <configuration>
        <system>
          <login/>
        </system>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

For more information about configuration information requests, see "Requesting Configuration Data Using NETCONF" on page 213. For a summary of the available 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 NETCONF

protocol defines the **<edit-config>** and **<copy-config>** operations 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>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <system>
          <login>
            <user>
              <name>admin</name>
              <full-name>Administrator</full-name>
              <class>superuser</class>
            </user>
          </login>
          <login/>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

For more information about configuration change requests, see "Editing the Configuration Using NETCONF" on page 139. For a summary of Junos XML configuration tag elements, see the *Junos XML API Configuration Developer Reference* and the XML API Explorer.

Related Documentation

- Generating Well-Formed XML Documents on page 31
- Parsing the NETCONF Server Response on page 52
- XML and NETCONF XML Management Protocol Conventions Overview on page 11
- <rpc> on page 94

Parsing the NETCONF Server Response

In a NETCONF session with a device running Junos OS, a client application sends RPCs to the NETCONF server to request information from and manage the configuration on the device. The NETCONF 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="urn:ietf:params:xml:ns:netconf:base:1.0" \
                xmlns:junos="http://xml.juniper.net/junos/release/junos" \
```

```
                [echoed attributes]>
    <!-- tag elements representing a response -->
</rpc-reply>
]]>]]>
```

The **xmlns** attribute in the opening **<rpc-reply>** tag defines the namespace for enclosed tag elements that do not have the **junos:** prefix in their names and that are not enclosed in a child container tag that has the **xmlns** attribute with a different value.

NOTE: Beginning in Junos OS Release 15.1, if you configure the **rfc-compliant** statement on the device, the NETCONF server explicitly declares the NETCONF namespace, which is bound to the **nc** prefix, and qualifies all NETCONF tags in its replies with the prefix.

The **xmlns:junos** attribute defines the default namespace for 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 NETCONF server device, for example 19.2R1.

Client applications must include code for parsing the stream of response tag elements coming from the NETCONF server, either processing them as they arrive or storing them until the response is complete. The NETCONF server returns three classes of responses:

- Operational Responses on page 53
- Configuration Information Responses on page 54
- Configuration Change Responses on page 54

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 NETCONF server returns the information requested by the **<get-interface-information>** tag in a response tag element called **<interface-information>**, and returns the information requested by the **<get-chassis-inventory>** tag in a 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 NETCONF Session" on page 205.

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 is for interface information for Junos OS Release 19.2. The opening tags appear on two lines here for legibility only:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"\
               xmlns:junos="http://xml.juniper.net/junos/19.2R1/junos">
```

```
    <interface-information \
                xmlns="http://xml.juniper.net/junos/19.2R1/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 the contents of operational response tag elements, see "Requesting Operational Information Using NETCONF" on page 203. 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.

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. The opening **<rpc-reply>** tag appears on two lines for legibility only. For information about the attributes in the opening **<configuration>** tag, see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"\
  xmlns:junos="http://xml.juniper.net/junos/19.2R1/junos">
  <data>
    <configuration attributes>
      <system>
        <login>
          <user>
            <name>admin</name>
            <full-name>Administrator</full-name>
            <!-- other data tag elements for the admin user -->
          </user>
        </login>
      </system>
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

Configuration Change Responses

Configuration change responses are responses to requests that change the state or contents of the device configuration. The NETCONF server indicates successful execution of a request by returning the **<ok/>** tag within the **<rpc-reply>** tag element:

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

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

If the operation fails, the **<rpc-reply>** tag element instead encloses an **<rpc-error>** element that describes the cause of the failure. For information about handling errors, see "Handling an Error or Warning in a NETCONF Session" on page 57.

Release History Table

Release	Description
15.1	Beginning in Junos OS Release 15.1, if you configure the **rfc-compliant** statement on the device, the NETCONF server explicitly declares the NETCONF namespace, which is bound to the **nc** prefix, and qualifies all NETCONF tags in its replies with the prefix.

Related Documentation

- Using a Standard API to Parse Response Tag Elements in NETCONF and Junos XML Protocol Sessions on page 55
- Handling an Error or Warning in a NETCONF Session on page 57
- Configuring RFC-Compliant NETCONF Sessions on page 66
- XML and NETCONF XML Management Protocol Conventions Overview on page 11
- Generating Well-Formed XML Documents on page 31
- <rpc-error> on page 94

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/~tjmather/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*
- Parsing the NETCONF Server Response on page 52

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&amp;#195;&amp;#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 a NETCONF Session

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

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <rpc-error>
    <error-severity>error-severity</error-severity>
    <error-path>error-path</error-path>
    <error-message>error-message</error-message>
    <error-info>
      <bad-element>command-or-statement</bad-element>
    </error-info>
  <rpc-error>
</rpc-reply>
]]>]]>
```

<bad-element> identifies the command or configuration statement that was being processed when the error or warning occurred. For a configuration statement, the **<error-path>** tag element enclosed in the **<rpc-error>** tag element specifies the statement's parent hierarchy level.

<error-message> describes the error or warning in a natural-language text string.

<error-path> specifies the path to the Junos OS configuration hierarchy level at which the error or warning occurred, in the form of the CLI configuration mode banner.

<error-severity> indicates the severity of the event that caused the NETCONF server to return the **<rpc-error>** tag element. The two possible values are **error** and **warning**.

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 an **<edit-config>** tag element

Client applications must be prepared to receive and handle an **<rpc-error>** tag element 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.

When the **<error-severity>** tag element has the value **error**, the usual response is for the client application to discard the information and terminate. When the **<error-severity>** tag element has the value **warning**, indicating that the problem is less serious, the usual response is for the client application to log the warning or pass it to the user and to continue parsing the server's response.

NOTE: Starting in Junos OS Release 18.2R2, 18.3R2, and 18.4R1, when you configure the **rfc-compliant** statement at the **[edit system services netconf]** hierarchy level to enforce certain behaviors by the NETCONF server, the NETCONF server cannot return an RPC reply that includes both an **<rpc-error>** element and an **<ok/>** element. If the operation is successful, but the server reply would include one or more **<rpc-error>** elements with a severity level of warning in addition to the **<ok/>** element, then the warnings are omitted.

Related Documentation

- Parsing the NETCONF Server Response on page 52
- <rpc-error> on page 94

Locking and Unlocking the Candidate Configuration Using NETCONF

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. This is equivalent to the CLI **configure exclusive** 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 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.

For more information about locking and unlocking the candidate configuration, see the following sections:

- Locking the Candidate Configuration on page 59
- Unlocking the Candidate Configuration on page 60

Locking the Candidate Configuration

To lock the candidate configuration, a client application emits the **<lock>** and **<target>** tag elements and the **<candidate/>** tag in the **<rpc>** tag element.

```
<rpc>
  <lock>
    <target>
      <candidate/>
    </target>
  </lock>
</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 NETCONF server confirms that it has locked the candidate by returning the **<ok/>** tag in the **<rpc-reply>** tag element.

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

If the NETCONF server cannot lock the configuration, the **<rpc-reply>** tag element instead encloses an **<rpc-error>** tag 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 NETCONF session identifier of the user or application. If the client application has the necessary Junos OS access privilege, it can terminate the session that holds the lock. For more information, see "Terminating a NETCONF Session" on page 60.
- The candidate configuration already includes changes that have not yet been committed. To commit the changes, see "Committing the Candidate Configuration Using NETCONF" on page 174. To discard uncommitted changes, see "Rolling Back Uncommitted Changes in the Candidate Configuration Using NETCONF" on page 155.

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. The lock persists until either the NETCONF session ends or the client application unlocks the configuration by emitting the **<unlock>** tag element, as described in "Unlocking the Candidate Configuration" on page 60.

If the candidate configuration is not committed before the client application unlocks it, or if the NETCONF 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>** and **<target>** tag elements and the **<candidate/>** tag in an **<rpc>** tag element.

```
<rpc>
  <unlock>
    <target>
      <candidate/>
    </target>
  </unlock>
</rpc>
]]>]]>
```

The NETCONF server confirms that it has unlocked the candidate by returning the **<ok/>** tag in the **<rpc-reply>** tag element.

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

If the NETCONF server cannot unlock the configuration, the **<rpc-reply>** tag element instead encloses an **<rpc-error>** tag element explaining the reason for the failure.

Related Documentation

- Understanding the Client Application's Role in a NETCONF Session on page 30
- <lock> on page 88
- <target> on page 96
- <unlock> on page 89

Terminating a NETCONF Session

In a NETCONF 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 NETCONF 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">
    <rpc-error>
        <error-severity>error</error-severity>
        <error-message>
            configuration database locked by:
user terminal (pid PID) on since YYYY-MM-DD hh:mm:ss TZ, idle hh:mm:ss
            exclusive
        </error-message>
    </rpc-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>** tag element. The **<session-id>** element specifies the PID obtained from the error message:

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

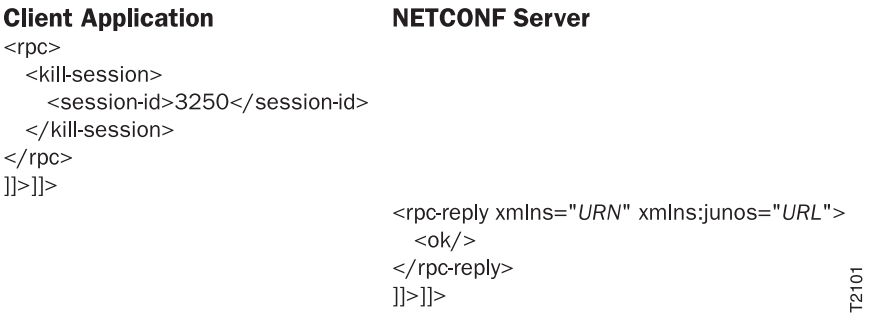
The NETCONF server confirms that it has terminated the other session by returning the **<ok/>** tag in the **<rpc-reply>** tag element:

```
<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 NETCONF 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 NETCONF 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 NETCONF" on page 175.

The following example shows how to terminate another session:



Related Documentation

- Ending a NETCONF Session and Closing the Connection on page 62
- Locking and Unlocking the Candidate Configuration Using NETCONF on page 58
- <kill-session> on page 88

Ending a NETCONF Session and Closing the Connection

When a client application is finished making requests, it ends the NETCONF session by emitting the empty **<close-session/>** tag within an **<rpc>** tag element:

```
<rpc>
  <close-session/>
</rpc>
]]>]]>
```

In response, the NETCONF server emits the **<ok/>** tag enclosed in an **<rpc-reply>** tag element:

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

Because the connection to the NETCONF server is an SSH subsystem, it closes automatically when the NETCONF session ends.

Related Documentation

- <close-session/> on page 79

Sample NETCONF Session

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

- Exchanging Initialization Tag Elements on page 63
- Sending an Operational Request on page 63
- Locking the Configuration on page 64

Exchanging Initialization Tag Elements

After the client application establishes a connection to a NETCONF server, the two exchange **<hello>** tag elements, as shown in the following example. For legibility, the example places the client application's **<hello>** tag element below the NETCONF server's. The two parties can actually emit their **<hello>** tag elements at the same time. For information about the **]]>]]>** character sequence used in this and the following examples, see "Generating Well-Formed XML Documents" on page 31. For a detailed discussion of the **<hello>** tag element, see "Exchanging <hello> Tag Elements" on page 46.

NETCONF Client Application
Server

```
<hello>
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file </capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
  </capabilities>
  <session-id>3911</session-id>
</hello>
]]>]]>
```

```
<hello>
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file</capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
  </capabilities>
</hello>
]]>]]>
```

T2102

Sending an Operational Request

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

Client Application

```
<rpc>
  <get-chassis-inventory>
    <detail/>
  </get-chassis-inventory>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" 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>
]]>]]>
```

T2103

Locking the Configuration

The client application then prepares to incorporate a change into the candidate configuration by emitting the **<lock/>** 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 NETCONF server returns an **<ok/>** tag in an **<rpc-reply>** tag element. For more information about locking the configuration, see "Locking and Unlocking the Candidate Configuration Using NETCONF" on page 58.

Client Application

```
<rpc>
  <lock>
    <target>
      <candidate/>
    </target>
  </lock>
</rpc>
]]>]]>
```

NETCONF Server

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

T2104

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. To confirm that the load operation was successful, the NETCONF server returns an **<ok/>** tag in an **<rpc-reply>** tag element.

Client Application **NETCONF Server**

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <system>
          <login>
            <class>
              <name>network-mgmt</name>
              <permissions>configure</permissions>
              <permissions>snmp</permissions>
              <permissions>system</permissions>
            </class>
          </login>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

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

T2105

Committing the Configuration

The client application then commits the candidate configuration. To confirm that the commit operation was successful, the NETCONF server returns an **<ok/>** tag in an **<rpc-reply>** tag element. For more information about the commit operation, see "Committing the Candidate Configuration Using NETCONF" on page 174.

Client Application **NETCONF Server**

```
<rpc>
  <commit/>
</rpc>
]]>]]>
```

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

T2106

Unlocking the Configuration

The client application unlocks (and by implication closes) the candidate configuration. To confirm that the unlock operation was successful, the NETCONF server returns an **<ok/>** tag in an **<rpc-reply>** tag element. For more information about unlocking a configuration, see "Locking and Unlocking the Candidate Configuration Using NETCONF" on page 58.

Client Application

```
<rpc>
  <unlock>
    <target>
      <candidate/>
    </target>
  </unlock>
</rpc>
]]>]]>
```

NETCONF Server

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

T2107

Closing the NETCONF Session

The client application closes the NETCONF session by emitting the **<close-session>** tag. For more information about closing the session, see "Ending a NETCONF Session and Closing the Connection" on page 62.

Client Application

```
<rpc>
  <close-session/>
</rpc>
]]>]]>
```

NETCONF Server

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

T2108

Related Documentation

- Generating Well-Formed XML Documents on page 31
- Starting the NETCONF Session on page 46
- Locking and Unlocking the Candidate Configuration Using NETCONF on page 58
- Ending a NETCONF Session and Closing the Connection on page 62

Configuring RFC-Compliant NETCONF Sessions

When using NETCONF to manage devices running Junos OS, you can require that the NETCONF server enforce certain behaviors that are compliant with RFC 4741, *NETCONF Configuration Protocol* during the NETCONF session by configuring the **rfc-compliant** statement at the **[edit system services netconf]** hierarchy level. Configuring the **rfc-compliant** statement affects the following aspects of the NETCONF session:

- Namespaces emitted in NETCONF server replies
- **<get>** and **<get-config>** elements in RPC replies in cases where there is no configuration data to return
- NETCONF server replies that would return both an **<ok/>** element and an **<rpc-error>** element with a severity level of warning

The differences are described in detail in the following sections:

Namespaces

In a NETCONF session with a device running Junos OS, the NETCONF server, by default, sets the default namespace to the NETCONF namespace in the opening tag of the server's reply, and NETCONF tag names are not qualified. For example:

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    ...
  </capabilities>
  <session-id>27700</session-id>
<hello>
```

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
```

When the **rfc-compliant** statement is configured on the device, the NETCONF server does not define a default namespace in its replies. Instead, the server includes a namespace declaration for the NETCONF namespace, which is bound to the **nc** prefix, and qualifies all NETCONF tags in its replies with the prefix. If you set the default namespace to the NETCONF namespace in an RPC request, the server discards the default namespace and emits its reply using only the declared namespace that is bound to the **nc** prefix.

The following sample output shows the NETCONF server's **<hello>** message and capabilities exchange when the **rfc-compliant** statement is configured. The **<hello>** tag contains the **xmlns:nc** declaration, and all NETCONF tags include the **nc** prefix.

```
<nc:hello xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <nc:capabilities>
    <nc:capability>urn:ietf:params:netconf:base:1.0</nc:capability>
    ...
  </nc:capabilities>
  <nc:session-id>27703</nc:session-id>
</nc:hello>
```

The following output shows a sample RPC reply when the **rfc-compliant** statement is configured:

```
<nc:rpc-reply
    xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
    xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
  <nc:data>
    <configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm"
```

```
          junos:changed-seconds="1417554471"
          junos:changed-localtime="2014-12-02 13:07:51 PST">
        <!--configuration data-->
    </configuration>
    <database-status-information>
      <database-status>
        <user>root</user>
        <terminal></terminal>
        <pid>47868</pid>
       <start-time junos:seconds="1417560303">2014-12-02 14:45:03 PST</start-time>

        <edit-path></edit-path>
      </database-status>
    </database-status-information>
  </nc:data>
</nc:rpc-reply>
```

Starting with Junos OS Release 17.2R1, when you configure the **rfc-compliant** statement and request configuration data in a NETCONF session, the server sets the default namespace for the **<configuration>** element to the same namespace as in the corresponding YANG model.

```
<rpc>
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>
]]>]]>

<nc:rpc-reply
  xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
  xmlns:junos="http://xml.juniper.net/junos/17.2R1/junos">
<nc:data>
<configuration
  xmlns="http://yang.juniper.net/yang/1.1/jc/configuration/junos/17.2R1.13"
  junos:commit-seconds="1493761452"
  junos:commit-localtime="2017-05-02 14:44:12 PDT"
  junos:commit-user="user">
  ...
</configuration>
</nc:data>
</nc:rpc-reply>
]]>]]>
```

Changes to <get> and <get-config> Operations

The **rfc-compliant** statement affects the **<get>** and **<get-config>** server replies in cases where there is no configuration data to return. This can occur, for example, when you

apply a filter to return a subset of the configuration, and that portion of the configuration is empty.

If you execute the **<get>** or **<get-config>** operation, and there is no configuration data in the requested hierarchy, then if the **rfc-compliant** statement is not configured, the RPC reply contains an empty **<configuration>** element inside the **<data>** element.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/15.1D0/junos">
<data>
<configuration>
</configuration>
</data>
</rpc-reply>
```

If you execute the **<get>** or **<get-config>** operation, and there is no configuration data in the requested hierarchy, then if the **rfc-compliant** statement is configured, the RPC reply returns an empty **<data>** element and omits the **<configuration>** element.

```
<nc:rpc-reply xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
<nc:data>
</nc:data>
</nc:rpc-reply>
```

<rpc-error> Elements with a Severity Level of Warning in RPC Replies

Starting in Junos OS Release 18.2R2, 18.3R2, and 18.4R1, when you configure the **rfc-compliant** statement, the NETCONF server cannot return an RPC reply that includes both an **<rpc-error>** element and an **<ok/>** element. If the operation is successful, but the server reply would include one or more **<rpc-error>** elements with a severity level of warning in addition to the **<ok/>** element, then the warnings are omitted. In earlier releases, or when the **rfc-compliant** statement is not configured, the NETCONF server might issue an RPC reply that includes both an **<rpc-error>** element with a severity level of warning and an **<ok/>** element.

For example, in earlier releases, or if the **rfc-compliant** statement is not configured, a commit operation might be successful but return a warning as in the following NETCONF server reply:

```
<nc:rpc-reply xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/17.4R1/junos">
    <nc:rpc-error>
        <nc:error-severity>warning</nc:error-severity>
        <nc:error-message>
            uid changed for jadmin (2001-&gt;2014)
        </nc:error-message>
    </nc:rpc-error>
    <nc:ok/>
</nc:rpc-reply>
]]>]]>
```

When the **rfc-compliant** statement is configured in the affected releases, the warning is omitted.

```
<nc:rpc-reply xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/18.4R1/junos">
<nc:ok/>
</nc:rpc-reply>
]]>]]>
```

Release History Table

Release	Description
18.4R1	Starting in Junos OS Release 18.2R2, 18.3R2, and 18.4R1, when you configure the **rfc-compliant** statement, the NETCONF server cannot return an RPC reply that includes both an **<rpc-error>** element and an **<ok/>** element.

Related Documentation

- rfc-compliant on page 420

CHAPTER 5

NETCONF Tracing Operations

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 ***size*k** to specify KB, ***size*m** to specify MB, or ***size*g** 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 72
- netconf on page 417
- ssh (NETCONF) on page 421
- traceoptions (NETCONF and Junos XML Protocol) on page 422

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

- 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 outcoming 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 71
- netconf on page 417
- ssh (NETCONF) on page 421
- traceoptions (NETCONF and Junos XML Protocol) on page 422

CHAPTER 6

NETCONF Protocol Operations

<close-session/>

Usage

```
<rpc>
  <close-session/>
</rpc>
]]>]]>
```

Description Request that the NETCONF server end the current session.

Related Documentation

- Ending a NETCONF Session and Closing the Connection on page 62
-]]>]]> on page 91
- <rpc> on page 94

<commit>

Usage

```
<rpc>
  <commit/>
</rpc>
]]>]]>
```

```
<rpc>
  <commit>
    <confirmed/>
    <confirm-timeout>rollback-delay</confirm-timeout>
  </commit>
</rpc>
]]>]]>
```

Description Request that the NETCONF server perform one of the variants of the commit operation on the candidate configuration or open configuration database:

- To commit the configuration immediately, making it the active configuration on the device, emit the empty **<commit/>** tag.

- To commit the configuration but require an explicit confirmation for the commit to become permanent, enclose the **<confirmed/>** tag in the **<commit>** tag element.

  NOTE: The **<confirmed/>** tag is not supported when committing configuration data to the ephemeral configuration database.

  By default, the NETCONF server rolls back to the previous running configuration after 600 seconds (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>** tag element) again 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 600 seconds (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/>** tag before the rollback deadline passes. The NETCONF server commits the candidate configuration and cancels the rollback. If the candidate configuration is still the same as the running configuration, the effect is the same as recommitting the current running configuration.

Contents

<confirmed>—Request a temporary commit of the candidate configuration. The device reverts to the previous active configuration after a specified time.

<confirm-timeout>—Specify the number of seconds before the device reverts to the previously active configuration. If this tag element is omitted, the default is 600 seconds (10 minutes).

Related Documentation

- Committing the Candidate Configuration Using NETCONF on page 174
- Committing the Candidate Configuration Only After Confirmation Using NETCONF on page 175

<copy-config>

Usage

```
<rpc>
  <copy-config>
    <target>
      <candidate/>
    </target>
    <source>
      <url format="(xml | text)">
        <!-- location specifier for file containing the new configuration -->
      </url>
    </source>
  </copy-config>
</rpc>
]]>]]>
```

Description Replace the entire existing candidate configuration or open configuration database with the configuration data contained in a file.

If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing a **<copy-config>** operation on the target **<candidate/>**, Junos OS performs the operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

Contents

<source>—Enclose the **<url>** tag element, which specifies the source of the configuration data.

<url>—Specify the file that contains the new configuration data to substitute for the data in the existing candidate configuration or open configuration database.

When the configuration data is formatted as Junos XML tag elements, set the **<url> format** attribute to "xml" or omit the attribute. When the configuration data is formatted as CLI configuration statements, set the **<url> format** attribute to "text". For more information, see "Uploading and Formatting Configuration Data in a NETCONF Session" on page 141.

The **<target>** tag element and its contents are explained separately.

Related Documentation

- Replacing the Candidate Configuration Using NETCONF on page 151
- <target> on page 96

<delete-config>

Usage

```
<rpc>
  <delete-config>
    <target>
      <candidate/>
    </target>
  </delete-config>
</rpc>
]]>]]>
```

Description Delete all configuration data in the existing 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 **<delete-config>** operation on the target **<candidate/>**, Junos OS performs the **<delete-config>** operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

Contents The **<target>** tag element and its contents are explained separately.

Related Documentation

- Deleting the Configuration Using NETCONF on page 156
- Deleting Configuration Elements Using NETCONF on page 161
- <target> on page 96

<discard-changes/>

Usage

```
<rpc>
  <discard-changes/>
</rpc>
]]>]]>
```

Description

Discard changes made to the candidate configuration and make its contents match the contents of the current running (active) configuration. This operation is equivalent to the Junos OS CLI configuration mode **rollback 0** command.

NOTE: The <discard-changes/> operation cannot be used to discard uncommitted changes that have been loaded into the ephemeral configuration database.

Related Documentation

- Rolling Back Uncommitted Changes in the Candidate Configuration Using NETCONF on page 155

<edit-config>

Usage

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>

    <!-- EITHER -->

        <config>
            <configuration>
                <!-- tag elements representing the data to incorporate -->
            </configuration>
        </config>

    <!-- OR -->

        <config-text>
            <configuration-text>
                <!-- configuration data in text format -->
            </configuration-text>
        </config-text>
```

```
    <!-- OR -->

        <url format="(xml | text)">
            <!-- location specifier for file containing data -->
        </url>

        <default-operation>(merge | none | replace)</default-operation>
        <error-option>(ignore-error | stop-on-error)</error-option>
        <test-option>(set | test-then-set)</test-option>
    </edit-config>
</rpc>
]]>]]>
```

Description Request that the NETCONF server incorporate configuration data into the candidate configuration or open configuration database. Provide the data in one of three ways:

- Include the **<config>** tag element to provide a data stream of Junos XML configuration tag elements to incorporate. The tag elements are enclosed in the **<configuration>** tag element.
- Include the **<config-text>** tag element to provide a data stream of CLI configuration statements to incorporate. The configuration statements are enclosed in the **<configuration-text>** tag element.
- Include the **<url>** tag element to specify the location of a file that contains the Junos OS configuration to incorporate. The format of the configuration data can be Junos XML elements or CLI configuration statements.

If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing the **<edit-config>** operation on the target **<candidate/>**, Junos OS performs the **<edit-config>** operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

Contents **<config>**—Enclose the **<configuration>** tag element.

<configuration>—Enclose the configuration data written in Junos XML. This configuration data is provided as a data stream and is incorporated into the candidate configuration or open configuration database. For information about the syntax for representing the elements to create, delete, or modify, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 19.

<config-text>—Enclose the **<configuration-text>** tag element.

<configuration-text>—Enclose the configuration data formatted as CLI configuration statements. This configuration data is provided as a data stream and is incorporated into the candidate configuration or open configuration database.

<default-operation>—(Optional) Specify how to incorporate the new configuration data into the candidate configuration or open configuration database, particularly when there are conflicting statements. The following are acceptable values:

- **merge**—Combine the new configuration data with the existing configuration according to the rules defined in "Setting the Edit Configuration Mode in a NETCONF Session" on page 147. This is the default mode if the **<default-operation>** tag element is omitted. It applies to all elements in the new data that do not have the **operation** attribute in their opening container tag to specify a different mode.
- **none**—Retain each configuration element in the existing configuration unless the new data includes a corresponding element that has the **operation** attribute in its opening container tag to specify an incorporation mode. This mode prevents the NETCONF server from creating parent hierarchy levels for an element that is being deleted. For more information, see "Setting the Edit Configuration Mode in a NETCONF Session" on page 147.
- **replace**—Discard the existing configuration data in the candidate configuration or open configuration database and replace it with the new data. For more information, see "Replacing the Candidate Configuration Using NETCONF" on page 151.

<error-option>—(Optional) Specify how the NETCONF server handles errors encountered while it incorporates the configuration data. The following are acceptable values:

- **ignore-error**—Specify that the NETCONF server continue to incorporate the new configuration data even if it encounters an error.
- **stop-on-error**—Specify that the NETCONF server stop incorporating the new configuration data when it encounters an error. This is the default behavior if the **<error-option>** tag element is omitted.

<test-option>—(Optional) Specify whether the NETCONF server validate the configuration data before incorporating it into the candidate configuration. The acceptable values defined in the NETCONF specification are **set** (no validation) and the default **test-then-set** (do not incorporate data if validation fails).

Regardless of the value provided, the NETCONF server for the Junos OS performs a basic syntax check on the configuration data in the **<edit-config>** tag element. It performs a complete syntactic and semantic validation on the candidate configuration in response to the **<validate>** and **<commit>** tag elements, but not for the **<edit-config>** tag element.

NOTE: **The <test-option> element is not supported when incorporating configuration data into the ephemeral configuration database.**

<url>—Specify the full pathname of the file that contains the configuration data to load. When the configuration data is formatted as Junos XML tag elements, set the **<url> format** attribute to "xml" or omit the attribute. When the configuration data is formatted as CLI configuration statements, set the **<url>** format attribute to "text".

For more information, see "Uploading and Formatting Configuration Data in a NETCONF Session" on page 141.

The **<target>** tag element and its contents are explained separately.

Related Documentation

- Changing Individual Configuration Elements Using NETCONF on page 156
- Editing the Configuration Using NETCONF on page 139
- Replacing the Candidate Configuration Using NETCONF on page 151
- Setting the Edit Configuration Mode in a NETCONF Session on page 147
- Uploading and Formatting Configuration Data in a NETCONF Session on page 141
- <target> on page 96

<get>

Usage

```
<rpc>
  <get [format="(json | set | text | xml)"]>
    <filter type="subtree">
      <configuration>
        <!-- tag elements representing the configuration elements to return -->
      </configuration>
    </filter>
  </get>
</rpc>
]]>]]>
```

Description Request the committed configuration and device state information from the NETCONF server. To display one or more sections of the configuration hierarchy (hierarchy levels or configuration objects), enclose the appropriate child tag elements in the **<filter>** element.

Attributes **format**—Specify the return format for the configuration data. If you omit this attribute, the server returns the configuration data formatted as Junos XML elements. Acceptable values are:

- **json**—Configuration statements are formatted in JavaScript Object Notation (JSON). Starting in Junos OS Release 14.2, you can display the configuration and device state information in JSON format.
- **set**—Configuration statements 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.

Contents

<filter>—(Optional) Enclose the **<configuration>** tag element. The optional **type** attribute indicates the kind of syntax used to represent the requested configuration elements; the only acceptable value is **subtree**.

To specify the configuration elements to return, include within the **<filter>** tag element the Junos XML tag elements that represent all levels of the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to each element to display. For information about the configuration elements available in the current version of the Junos OS, see *Junos XML API Configuration Developer Reference*.

Release History Table

Release	Description
14.2	Starting in Junos OS Release 14.2, you can display the configuration and device state information in JSON format.

Related Documentation

- Requesting the Committed Configuration and Device State Using NETCONF on page 211

<get-config>

Usage

```
<rpc>
  <get-config>
    <source>
      <( candidate | running )/>
    </source>
  </get-config>

  <get-config>
    <source>
      <( candidate | running )/>
    </source>
    <filter type="subtree">
      <configuration>
        <!-- tag elements for each configuration element to return -->
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

Description Request configuration data from the NETCONF server. The child tag elements **<source>** and **<filter>** specify the source and scope of data to display:

- To display the entire active configuration, enclose the **<source>** tag element and **<running/>** tag in the **<get-config>** tag element.
- To display either the entire candidate configuration or all configuration data in the open configuration database, enclose the **<source>** tag element and **<candidate/>** tag in the **<get-config>** tag element.

  If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing the **<get-config>** operation, setting the source to **<candidate/>** retrieves the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration.
- To display one or more sections of the configuration hierarchy (hierarchy levels or configuration objects), enclose the appropriate child tag elements in the **<source>** and **<filter>** tag elements.

Contents

<candidate/>—Specify the open configuration database, or if there is no open database, the candidate configuration.

<configuration>—Enclose tag elements that specify which configuration elements to return.

<filter>—Enclose the **<configuration>** tag element. The mandatory **type** attribute indicates the kind of syntax used to represent the requested configuration elements; the only acceptable value is **subtree**.

To specify the configuration elements to return, include within the **<filter>** tag element the Junos XML tag elements that represent all levels of the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to each element to display. For information about the configuration elements available in the current version of the Junos OS, see the XML API Explorer.

<running/>—Specify the active (mostly recently committed) configuration.

<source>—Enclose the tag that specifies the source of the configuration data. To specify either the candidate configuration or an open configuration database, include the **<candidate/>** tag. To specify the active configuration, include the **<running/>** tag.

Usage Guidelines See "Requesting Configuration Data Using NETCONF" on page 213.

Related Documentation

- <data> on page 91

<kill-session>

Usage

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

Description

Request that the NETCONF server terminate another CLI or NETCONF 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.

The client application must have the Junos OS **maintenance** permission to perform this operation.

Contents

<session-id>—Process identifier (PID) of the entity conducting the session to terminate. The PID is reported in the **<rpc-error>** tag element that the NETCONF server generates when it cannot lock a configuration as requested.

Related Documentation

- Terminating a NETCONF Session on page 60
- <lock> on page 88
- <rpc-error> on page 94

<lock>

Usage

```
<rpc>
  <lock>
    <target>
      <candidate/>
    </target>
  </lock>
</rpc>
]]>]]>
```

Description

Request that the NETCONF server 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 client application must emit the **<unlock/>** tag to unlock the configuration.

If the NETCONF session ends or the application emits the **<unlock>** tag element before the candidate configuration is committed, all changes made to the candidate are discarded.

Contents The **<target>** tag element and its contents are explained separately.

Related Documentation
- Locking and Unlocking the Candidate Configuration Using NETCONF on page 58
- <rpc> on page 94
- <target> on page 96
- <unlock> on page 89

<unlock>

Usage

```
<rpc>
  <unlock>
    <target>
      <candidate/>
    </target>
  </unlock>
</rpc>
]]>]]>
```

Description Request that the NETCONF server unlock and close the candidate configuration, which the client application previously locked by emitting the **<lock>** tag element. Until the application emits this tag element, other users or applications can read the configuration but cannot change it.

Contents The **<target>** tag element and its contents are explained separately.

Related Documentation
- Locking and Unlocking the Candidate Configuration Using NETCONF on page 58
- <lock> on page 88
- <target> on page 96

<validate>

Usage

```
<rpc>
  <validate>
    <source>
      <candidate/>
    </source>
  </validate>
</rpc>
]]>]]>
```

Description Check that the candidate configuration is syntactically valid.

Contents

<source>—Enclose the tag that specifies the configuration to validate.

<candidate/>—Specify the candidate configuration.

Related Documentation

- Verifying the Candidate Configuration Syntax Using NETCONF on page 173

CHAPTER 7

NETCONF Request and Response Tags

]]>]]>

Usage

```
<hello>
  <!-- child tag elements included by client application or NETCONF server -->
</hello>
]]>]]>
```

```
<rpc [attributes]>
  <!-- tag elements in a request from a client application -->
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <!-- tag elements in the response from the NETCONF server -->
</rpc-reply>
]]>]]>
```

Description

Signal the end of each XML document sent by the NETCONF server and client applications. Client applications send the sequence after its closing **</hello>** tag and each closing **</rpc>** tag. The NETCONF server sends the sequence after its closing **</hello>** tag and each closing **</rpc-reply>** tag.

Use of this signal is required by RFC 4742, *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, available at http://www.ietf.org/rfc/rfc4742.txt .

Related Documentation

- Generating Well-Formed XML Documents on page 31
- <hello> on page 93
- <rpc> on page 94
- <rpc-reply> on page 95

<data>

Usage

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

```
  <data>
    <configuration>
      <!-- Junos XML tag elements for the configuration data -->
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

Description Encloses configuration data and device information returned by the NETCONF server in response to a **<get>** request or configuration data returned by the NETCONF server in response to a **<get-config>** request.

NOTE: The NETCONF server, by default, returns configuration data formatted as Junos XML tag elements. The configuration data enclosed in the **<data>** element can vary if a client application requests a different format in a **<get>** request.

Contents **<configuration>**—Encloses configuration tag elements. It is the top-level tag element in the Junos XML API, equivalent to the **[edit]** hierarchy level in the Junos OS CLI. For information about Junos OS configuration elements, see the *Junos XML API Configuration Developer Reference*.

Usage Guidelines See "Requesting Configuration Data Using NETCONF" on page 213.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Developer Reference*
- <get> on page 85
- <get-config> on page 86
- <rpc-reply> on page 95

<error-info>

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <rpc-error>
    <error-info>
      <bad-element>command-or-statement</bad-element>
    </error-info>
  </rpc-error>
</rpc-reply>
]]>]]>
```

Description Provides additional information about the event or condition that causes the NETCONF server to report an error or warning in the **<rpc-error>** tag element.

Contents

<bad-element>—Identifies the command or configuration statement that was being processed when the error or warning occurred. For a configuration statement, the **<error-path>** tag element enclosed in the **<rpc-error>** tag element specifies the statement's parent hierarchy level.

Related Documentation

- Handling an Error or Warning in a NETCONF Session on page 57
- <rpc-error> on page 94
- <rpc-reply> on page 95

<hello>

Usage

```
<!-- emitted by a client application -->
<hello>
  <capabilities>
    <capability>URI</capability>
  </capabilities>
</hello>
]]>]]>
```

```
<!-- emitted by the NETCONF server -->
<hello>
  <capabilities>
    <capability>URI</capability>
  </capabilities>
  <session-id>session-identifier</session-id>
</hello>
]]>]]>
```

Description

Specify which operations, or *capabilities*, the emitter supports from among those defined in the NETCONF specification. The client application must emit the **<hello>** tag element before any other tag element during the NETCONF session, and must not emit it more than once.

Contents

<capabilities>—Encloses one or more **<capability>** tags, which together specify the set of supported NETCONF operations.

<capability>—Specifies the uniform resource identifier (URI) of a capability defined in the NETCONF specification or by a vendor. Each capability from the NETCONF specification is represented by a uniform resource name (URN). Capabilities defined by vendors are represented by URNs or URLs.

<session-id>—(Generated by NETCONF server only) Specifies the UNIX process ID (PID) of the NETCONF server for the session.

Related Documentation

- Exchanging <hello> Tag Elements on page 46

<ok/>

Usage

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

Description Indicates that the NETCONF server successfully performed a requested operation that changes the state or contents of the device configuration.

Related Documentation

- Configuration Change Responses on page 54
- <rpc-reply> on page 95

<rpc>

Usage

```
<rpc [attributes]>]
  <!-- tag elements in a request from a client application -->
</rpc>
]]>]]>
```

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 NETCONF server echoes the attribute unchanged in its opening **<rpc-reply>** tag, making it simple to map the response to the initiating request. The NETCONF specification assigns the name **message-id** to this attribute.

Related Documentation

- Sending Requests to the NETCONF Server on page 50
- <rpc-reply> on page 95

<rpc-error>

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <rpc-error>
    <error-severity>error-severity</error-severity>
    <error-path>error-path</error-path>
    <error-message>error-message</error-message>
    <error-info>...</error-info>
  </rpc-error>
</rpc-reply>
```

```
]]>]]>
```

Description Indicate that the NETCONF 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 that 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 NETCONF server does not necessarily emit all child tag elements; it omits tag elements that are not relevant to the current request.

NOTE: Starting in Junos OS Release 18.2R2, 18.3R2, and 18.4R1, when you configure the **rfc-compliant** statement at the **[edit system services netconf]** hierarchy level to enforce certain behaviors by the NETCONF server, the NETCONF server cannot return an RPC reply that includes both an **<rpc-error>** element and an **<ok/>** element. If the operation is successful, but the server reply would include one or more **<rpc-error>** elements with a severity level of warning in addition to the **<ok/>** element, then the warnings are omitted.

Contents **<error-message>**—Describes the error or warning in a natural-language text string.

<error-path>—Specifies the path to the Junos OS configuration hierarchy level at which the error or warning occurred, in the form of the CLI configuration mode banner.

<error-severity>—Indicates the severity of the event that caused the NETCONF server to return the **<rpc-error>** tag element. The two possible values are **error** and **warning**.

The **<error-info>** tag element is described separately.

Related Documentation

- Handling an Error or Warning in a NETCONF Session on page 57
- <error-info> on page 92
- <rpc-reply> on page 95

<rpc-reply>

Usage

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <!-- tag elements in a reply from the NETCONF server-->
</rpc-reply>
]]>]]>
```

Description Encloses all tag elements in a reply from the NETCONF server. The immediate child tag element is usually one of the following:

- The Junos XML tag element that encloses the data requested by a client application with a Junos XML operational request tag element; for example, the **<interface-information>** tag element in response to the **<get-interface-information>** tag element
- The **<data>** tag element, to enclose the data requested by a client application with either the **<get>** or the **<get-config>** tag element
- The **<ok/>** tag, to confirm that the NETCONF server successfully performed an operation that changes the state or contents of a configuration (such as a lock, change, or commit operation)
- The **<output>** tag element, if the Junos XML API does not define a specific tag element for requested operational information
- The **<rpc-error>** tag element, if the requested operation generated an error or warning

Attributes **xmlns**—Name the default XML namespace for the enclosed tag elements.

Related Documentation

- Parsing the NETCONF Server Response on page 52
- <data> on page 91
- <ok/> on page 94
- <output> in the *Junos XML API Operational Developer Reference*
- <rpc> on page 94
- <rpc-error> on page 94

<target>

Usage

```
<rpc>
  <( copy-config | delete-config | edit-config | lock | unlock )>
    <target>
      <candidate/>
    </target>
  </( copy-config | delete-config | edit-config | lock | unlock )>
</rpc>
]]>]]>
```

Description Specify the configuration on which to perform an operation.

If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing a **<copy-config>**, **<delete-config>**, or **<edit-config>** operation on the target **<candidate/>**, Junos OS performs the requested operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration. Client applications can only perform the **<lock>** and **<unlock>** operations on the candidate configuration.

Contents

<candidate/>—Specify the configuration on which to perform the operation, either the open configuration database, or if there is no open database, the candidate configuration. This is the only acceptable value for Junos OS.

Related Documentation

- Deleting the Configuration Using NETCONF on page 156
- Editing the Configuration Using NETCONF on page 139
- Locking and Unlocking the Candidate Configuration Using NETCONF on page 58
- Replacing the Candidate Configuration Using NETCONF on page 151
- <copy-config> on page 80
- <delete-config> on page 81
- <edit-config> on page 82
- <lock> on page 88
- <unlock> on page 89

CHAPTER 8

Junos XML Protocol Elements Supported in NETCONF Sessions

<abort/>

Usage

```
<rpc>
  <!-- child tag elements -->
</rpc>
<abort/>
```

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

- *Halting a Request in Junos XML Protocol Sessions*
- <abort-acknowledgement/> on page 99

<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>*

<checksum-information>

Usage

```
<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>
```

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 Encloses tag elements that include the file to check, the checksum algorithm used, and the checksum output.

Contents

<checksum>—Resulting value from the checksum computation.

<computation-method>—Checksum algorithm used. Currently, all checksum computations use the MD5 algorithm; thus, the only possible value is MD5.

<file-checksum>—Wrapper that holds the resulting **<input-file>**, **<computation-method>**, and **<checksum>** attributes for a particular checksum computation.

<input-file>—Name and path of the file that the checksum algorithm was run against.

Related Documentation
- <get-checksum-information> on page 109

<close-configuration/>

Usage

```
<rpc>
  <close-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

Close the open configuration database and discard any uncommitted changes.

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 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*
- <open-configuration> on page 121
- <request-end-session/> on page 123

<commit-configuration>

Usage

```
<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>
        <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*
- *Committing a Private Copy of the Configuration Using the Junos XML Protocol*
- *Committing a Configuration at a Specified Time Using the Junos XML Protocol*
- *Committing the Candidate Configuration Only After Confirmation Using the Junos XML Protocol*
- *Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol*
- *Logging a Message About a Commit Operation Using the Junos XML Protocol*
- <commit-results> on page 105
- <open-configuration> on page 121

<commit-results>

Usage

```
<rpc-reply xmlns:junos="URL">
    <!-- for the candidate configuration or ephemeral configuration -->
    <commit-results>
        <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 **<commit-configuration>** request by a client application. The **<commit-results>** element contains information about the requested commit operation performed by the server on a particular Routing Engine.

Contents

<load-success/>—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 **<routing-engine>** tag element is described separately.

Related Documentation

- *Committing the Candidate Configuration Using the Junos XML Protocol*
- <commit-configuration> on page 101
- <routing-engine> on page 123

<commit-revision-information>

Usage

```
<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>
```

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.
Element introduced in Junos OS Release 16.1.

Description 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.

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, re0-1365168149-1), which has the following format:

```
<routing-engine-name>-<timestamp>-<counter>
```

Contents

<old-db-revision>—Indicates the old configuration revision identifier, which is the identifier of the configuration prior to the previously successfully committed configuration.

<new-db-revision>—Indicates the new configuration revision identifier, which is the identifier of the last successfully committed configuration.

<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.

Related Documentation

- <commit-results> on page 105
- <routing-engine> on page 123

<database-status>

Usage

```
<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>
```

Release Information

This is a Junos XML management protocol response tag. It is a Juniper Networks proprietary extension to NETCONF and is identified in the capabilities exchange by the

URI http://xml.juniper.net/netconf/junos/1.0 . This operation is only supported in NETCONF sessions on Juniper Networks devices running Junos OS.

Description Describes a user or NETCONF 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/>—Indicate that the user has scheduled a commit operation for a later time.

<edit-path>—Specify the user's current location in the configuration hierarchy, in the form of the CLI configuration mode banner.

<exclusive/>—Indicate 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>—Specify how much time has passed since the user last performed an operation in the database.

<pid>—Specify the process ID of the Junos OS management process (mgd) that is handling the user's login session.

<start-time>—Specify 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>—Identify the UNIX terminal assigned to the user's connection.

<user>—Specify the Junos OS login ID of the user whose login to the configuration database caused the error.

Related Documentation

- <database-status-information> on page 108
- <xnm:error> on page 125

<database-status-information>

Usage

```
<data>
    <database-status-information>
        <database-status>...</database-status>
    </database-status-information>
</data>
```

```
<xnm:error>
    <database-status-information>
        <database-status>...</database-status>
    </database-status-information>
</xnm:error>
```

Release Information This is a Junos XML management protocol response tag. It is a Juniper Networks proprietary extension to NETCONF and is identified in the capabilities exchange by the URI http://xml.juniper.net/netconf/junos/1.0 . This operation is only supported in NETCONF sessions on Juniper Networks devices running Junos OS.

Description Describes one or more users who have an open editing session in the configuration database.

The "<database-status>" on page 107 tag element is explained separately.

Related Documentation
- <database-status> on page 107
- <xnm:error> on page 125

<end-session/>

Usage

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

Related Documentation
- *Ending a Junos XML Protocol Session and Closing the Connection*
- <request-end-session/> on page 123

<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.
Command added in Junos OS Release 9.2R1.

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 100

<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.
interface-ranges attribute added in Junos OS Release 10.3R1.
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.

When you compare the candidate configuration to the active configuration (**rollback="0"**), you can display the differences in formats other than text by including the appropriate value for the **format** attribute in the request. You can display the differences in XML format starting in Junos OS Release 15.1R1, and you can display the differences in JSON format starting in Junos OS Release 16.1R1.

NOTE: Starting in Junos OS Release 16.2R2, when you compare the candidate and active configurations and display the differences in XML or JSON format, the device omits the <configuration> tag in the XML output and omits the configuration object in the JSON output if the comparison either returns no differences or if the comparison returns differences for only non-native configuration data, for example, configuration data associated with an OpenConfig data model.

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*
- *junos:changed*
- *junos:group*
- *junos:interface-range*
- *junos:key*
- *Junos XML API Configuration Developer Reference*

<load-configuration>

Usage

```
<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>
            <!-- 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: Staring in Junos OS Release 18.1R1, the ephemeral configuration database supports loading configuration data using the action attribute values of override and replace in addition to the previously supported 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 18.1R1 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 18.1R1 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*/%2F*path/filename***.

Related Documentation

- *Requesting Configuration Changes Using the Junos XML Protocol*
- <load-configuration-results> on page 119
- *replace*
- entries for **<configuration>**, **<configuration-text>**, and **<configuration-set>** in the *Junos XML API Configuration Developer Reference*

<load-configuration-results>

Usage

```
<rpc-reply xmlns:junos="URL">
    <load-configuration-results>
        <load-success/>
        <load-error-count>errors</load-error-count>
    </load-configuration-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 NETCONF or Junos XML protocol server in response to a **<load-configuration>** request by a client application.

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.

Contents

<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.

<load-success/>—Indicates that the server successfully loaded new data into the candidate configuration or open configuration database.

Related Documentation

- <load-configuration> on page 115

<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*

- <unlock-configuration/> on page 124

<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 115 and "<commit-configuration>" on page 101.

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.

NOTE: Starting in Junos OS Release 18.2R1, the Junos XML protocol **<open-configuration>** operation does not emit an "**uncommitted changes will be discarded on exit**" warning message when opening a private copy of the candidate configuration. However, Junos OS still discards the uncommitted changes upon closing the private copy.

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*.
- <close-configuration/> on page 101
- <commit-configuration> on page 101
- <lock-configuration/> on page 120

<reason>

Usage

```
<xnm:error | xnm:warning>
    <reason>
        <daemon>process</daemon>
        <process-not-configured/>
        <process-disabled/>
        <process-not-running/>
    </reason>
</xnm:error | xnm:warning>
```

Release Information

This is a Junos XML management protocol response tag. It is a Juniper Networks proprietary extension to NETCONF and is identified in the capabilities exchange by the URI http://xml.juniper.net/netconf/junos/1.0 . This operation is only supported in NETCONF sessions on Juniper Networks devices running Junos OS.

Description

Child element included in an **<xnm:error>** or **<xnm:warning>** element in a NETCONF 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

- <xnm:error> on page 125
- <xnm:warning> on page 126

<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 109

<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 105
- <commit-revision-information> on page 106

<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*
- <lock-configuration/> on page 120

<xnm:error>

Usage

```
<xnm:error xmlns="namespace-URL" xmlns:xnm="namespace-URL">
    <parse/>
    <source-daemon>module-name </source-daemon>
    <filename>filename</filename>
    <line-number>line-number </line-number>
    <column>column-number</column>
    <token>input-token-id </token>
    <edit-path>edit-path</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>
```

Release Information This is a Junos XML management protocol response tag. It is a Juniper Networks proprietary extension to NETCONF and is identified in the capabilities exchange by the URI http://xml.juniper.net/netconf/junos/1.0 . This operation is only supported in NETCONF sessions on Juniper Networks devices running Junos OS.

Description Indicates that the NETCONF 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 NETCONF server does not necessarily emit all child tag elements; it omits tag elements that are not relevant to the current request.

Attributes

xmlns—Names the XML namespace for the contents of the tag element. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where *version* is a string such as 1.1.

xmlns:xnm—Names the XML namespace for child tag elements that have the **xnm:** prefix on their names. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where *version* is a string such as 1.1.

Contents

<column>—(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

- <database-status-information> on page 108
- <reason> on page 122
- <xnm:warning> on page 126

<xnm:warning>

Usage

```
<xnm:warning xmlns="namespace-URL" xmlns:xnm="namespace-URL">
    <source-daemon>module-name </source-daemon>
    <filename>filename</filename>
    <line-number>line-number </line-number>
    <column>column-number</column>
    <token>input-token-id </token>
    <edit-path>edit-path</edit-path>
    <statement>statement-name </statement>
    <message>error-string</message>
    <reason>...</reason>
</xnm:warning>
```

Release Information

This is a Junos XML management protocol response tag. It is a Juniper Networks proprietary extension to NETCONF and is identified in the capabilities exchange by the URI http://xml.juniper.net/netconf/junos/1.0 . This operation is only supported in NETCONF sessions on Juniper Networks devices running Junos OS.

Description Indicates 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**—Names the XML namespace for the contents of the tag element. The value is a URL of the form **http://xml.juniper.net/xnm/*version*/xnm**, where ***version*** is a string such as 1.1.

xmlns:xnm—Names the XML namespace for child tag elements that have the **xnm:** prefix 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

- <reason> on page 122
- <xnm:error> on page 125

CHAPTER 9

Junos XML Protocol Element Attributes Supported in NETCONF Sessions

junos:changed-localtime

Usage

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

Description (Displayed when the candidate configuration is requested) Specifies the time when the configuration was last changed as the date and time in the device's local time zone.

Usage Guidelines See "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Developer Reference*
- <rpc-reply> on page 95
- junos:changed-seconds on page 129
- xmlns on page 134

junos:changed-seconds

Usage

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

Description (Displayed when the candidate configuration is requested) Specifies the time when the configuration was last changed as the number of seconds since midnight on 1 January 1970.

Usage Guidelines See "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Developer Reference*
- <rpc-reply> on page 95
- junos:changed-localtime on page 129
- xmlns on page 134

junos:commit-localtime

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration xmlns="URL" 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) Specifies the time when the configuration was committed as the date and time in the device's local time zone.

Usage Guidelines See "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Developer Reference*
- <rpc-reply> on page 95
- junos:commit-user on page 131
- junos:commit-seconds on page 130
- xmlns on page 134

junos:commit-seconds

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration xmlns="URL" junos:commit-seconds="seconds" \
    junos:commit-localtime="YYY-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) Specifies the time when the configuration was committed as the number of seconds since midnight on 1 January 1970.

Usage Guidelines See "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Developer Reference*
- <rpc-reply> on page 95
- junos:commit-user on page 131
- junos:commit-localtime on page 130
- xmlns on page 134

junos:commit-user

Usage

```
<rpc-reply xmlns:junos="URL">
  <configuration xmlns="URL" junos:commit-seconds="seconds" \
    junos:commit-localtime="YYY-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) Specifies the Junos OS username of the user who requested the commit operation.

Usage Guidelines See "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Developer Reference*
- <rpc-reply> on page 95
- junos:commit-localtime on page 130
- junos:commit-seconds on page 130
- xmlns on page 134

operation

Usage

```
<rpc>
  <edit-config>
    <config>
      <configuration>
        <!-- opening tags for each parent of the changing element -->
          <changing-element operation="(create | delete | replace)">
            <name>identifier</name>
            <!-- if changing element has an identifier -->
            <!-- other child tag elements, if appropriate -->
          </changing-element>
        <!-- closing tags for each parent of the changing element -->
      </configuration>
    </config>
    <!-- other child tag elements of the <edit-config> tag element -->
  <edit-config>
</rpc>
]]>]]>
```

Description

Specify how the NETCONF server incorporates an individual configuration element into the target configuration, which can be either the candidate configuration or the open configuration database. If the attribute is omitted, the element is merged into the configuration according to the rules defined in "Setting the Edit Configuration Mode in a NETCONF Session" on page 147. The following are acceptable values:

create—Create the specified element in the target configuration only if the element does not already exist.

delete—Delete the specified element from the target configuration. We recommend that the **<default-operation>** tag element with the value **none** also be included in the **<edit-config>** tag element.

replace—Replace the specified element in the target configuration with the provided new configuration data.

NOTE: The **operation="replace"** attribute is not supported when loading configuration data into the ephemeral configuration database.

Related Documentation

- Changing Individual Configuration Elements Using NETCONF on page 156
- Creating Configuration Elements Using NETCONF on page 159
- Deleting Configuration Elements Using NETCONF on page 161
- Replacing Configuration Elements Using NETCONF on page 167
- Setting the Edit Configuration Mode in a NETCONF Session on page 147
- **<configuration>** in the *Junos XML API Configuration Developer Reference*

- <edit-config> on page 82

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>
```

Release Information Attribute introduced in Junos OS Release 15.1R1.

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 168
- *Using Global Replace in the Junos OS Configuration*
- *Common Regular Expressions to Use with the replace Command*
- *replace*

xmlns

Usage

```
<rpc-reply xmlns:junos="URL">
    <operational-response xmlns="URL-for-DTD">
        <!-- Junos XML tag elements for the requested operational data -->
    </operational-response>
</rpc-reply>
<rpc-reply xmlns:junos="URL">
    <configuration xmlns="URL" junos:(changed | commit)-seconds="seconds" \
                junos:(changed | 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

For operational responses, defines 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.

For configuration data responses, define the XML namespace for the enclosed tag elements.

Usage Guidelines

See "Requesting Operational Information Using NETCONF" on page 203 and "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

Related Documentation

- **<configuration>** in the *Junos XML API Configuration Developer Reference*
- <rpc-reply> on page 95
- junos:changed-localtime on page 129
- junos:changed-seconds on page 129
- junos:commit-user on page 131
- junos:commit-localtime on page 130

- junos:commit-seconds on page 130

PART 3

Managing Configurations Using NETCONF

CHAPTER 10

Changing the Configuration Using NETCONF

Editing the Configuration Using NETCONF

In a NETCONF session with a device running Junos OS, you can use NETCONF XML management protocol operations along with Junos XML or command-line interface (CLI) configuration statements to change the configuration on a routing, switching, or security platform. The NETCONF protocol operations **<copy-config>**, **<edit-config>**, and **<discard-changes>** offer functionality that is analogous to configuration mode commands in the Junos OS CLI. The Junos XML tag elements described here correspond to Junos OS configuration statements.

To change the configuration on a device, a client application emits the **<copy-config>**, the **<edit-config>**, or the **<discard-changes>** tag element and the corresponding tag subelements within the **<rpc>** tag element.

The following examples shows the various tag elements available:

```
<rpc>
    <copy-config>
        <target><candidate/></target>
        <error-operation> (ignore-error | stop-on-error) </error-operation>
        <source><url>location</url></source>
    </copy-config>
</rpc>
]]>]]>
```

```
<rpc>
     <edit-config>
        <target><candidate/></target>
        <default-operation>operation</default-operation>
        <error-operation>error</error-operation>
        <(config | config-text | url)>
            <!-- configuration change file or data -->
        </(config | config-text | url)>
    </edit-config>
</rpc>
]]>]]>
```

```
<rpc>
    <discard-changes/>
</rpc>
]]>]]>
```

The only acceptable value for the **<target>** element is **<candidate/>**, which can refer to either the candidate configuration or the open configuration database. If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing a **<copy-config>** or **<edit-config>** operation, Junos OS performs the operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

The three tags—**<copy-config>**, **<edit-config>**, and **<discard-changes>**—correspond to the three basic configuration tasks available to you, which are described here:

- Overwriting the target configuration with a new configuration—Using the **<copy-config>** tag element, you can replace the configuration in the target configuration with a new configuration.
- Editing configuration elements—Using the **<edit-config>** tag element, you can add, change, or delete specific configuration elements within the target configuration. To specify how the device should handle configuration changes, see "Setting the Edit Configuration Mode in a NETCONF Session" on page 147.
- Rolling back changes to the current configuration—Using the **<discard-changes>** tag element, you can roll back the candidate configuration to match the contents of the current running (active) configuration. This tag element provides functionality analogous to the CLI command **rollback 0**.

NOTE: The <discard-changes/> tag element cannot be used to discard uncommitted changes that have been loaded into the ephemeral configuration database.

Related Documentation

- Uploading and Formatting Configuration Data in a NETCONF Session on page 141
- Setting the Edit Configuration Mode in a NETCONF Session on page 147
- Replacing the Candidate Configuration Using NETCONF on page 151
- Rolling Back Uncommitted Changes in the Candidate Configuration Using NETCONF on page 155
- Understanding the Client Application's Role in a NETCONF Session on page 30
- <copy-config> on page 80
- <discard-changes/> on page 82
- <edit-config> on page 82

Uploading and Formatting Configuration Data in a NETCONF Session

In a NETCONF session with a device running Junos OS, a client application can specify the delivery mechanism and the format of the configuration data used when delivering configuration changes to the device. Client applications can use a text file or streaming data to upload configuration data in one of the accepted formats to the candidate configuration or open configuration database.

A client can choose to stream configuration changes within the 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 in line, using your NETCONF 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 **<copy-config>** and **<edit-config>** operations accept one of two formats for the Junos OS configuration data: Junos XML or CLI configuration statements. The choice between one data format over the other is personal preference.

NOTE: When managing devices running Junos OS, a client application can use the Junos XML protocol <load-configuration> operation in a NETCONF session to upload configuration data formatted using JSON or configuration mode set commands, in addition to Junos XML or CLI configuration statement formats.

The delivery mechanism and the format are discussed in detail in the following sections:

Referencing Configuration Data Files

To upload configuration data stored in a file, a client application emits the file location between the **<url>** tags within the **<rpc>** and the **<edit-config>** or **<copy-config>** tag elements.

```
<rpc>
    <copy-config>
        <target>
            <candidate/>
        </target>
        <source>
            <url>
               <!-- location and name of file containing configuration data
-->
            </url>
        </source>
    </copy-config>
</rpc>
]]>]]>
```

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>
        <url>
           <!-- location and name of file containing configuration data -->
        </url>
    </edit-config>
</rpc>
]]>]]>
```

The data within these files can be formatted as either Junos XML elements or CLI configuration statements. When the configuration data is formatted as CLI configuration statements, include the **format="text"** attribute in the **<url>** tag.

```
<url format=”text”>
    <!-- location and name of file containing configuration data -->
</url>
```

The configuration file can be placed locally or as a network resource.

- When placed locally, the configuration file path can be relative or absolute:
  - Relative file path—The file location is based on the user's home directory.
  - Absolute file path—The file location is based on the directory structure of the device, for example ***<drive>:filename*** or ***<drive>/:path/filename***, If you are using removable media, the drive can be in the MS-DOS or UNIX (UFS) format.

- When located on the network, the configuration file can be accessed using FTP or HTTP:
  - FTP example:

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

    NOTE: The default value for the FTP *path* variable is the user's home directory. Thus, by default the file path to the configuration file is relative to the user directory. To specify an absolute path when using FTP, start the path with the characters %2F; for example: ftp://*username*:*password*@*hostname*/%2F*path*/*filename*.

  - HTTP example:

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

Before loading the file, the client application or an administrator saves Junos XML tag elements or CLI configuration statements as the contents of the file. The file includes the tag elements or configuration statements representing all levels of the configuration hierarchy from the root (represented by the **<configuration>** tag element) down to each element to change. The notation is the same as that used to request configuration information. For more detailed information about the Junos XML representation of Junos OS configuration statements, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 19.

The following example shows how to incorporate configuration data stored in the file **/var/tmp/configFile** on the FTP server called **ftp.myco.com**:

Client Application **NETCONF Server**

```
<rpc message-id="messageID">
  <edit-config>
    <target>
      <candidate/>
    </target>
    <url>
      ftp://admin:AdminPwd@ftp.myco.com/%F2var/tmp/configFile
    </url>
  </edit-config>
</rpc>
]]>]]>
```

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

T2134

Streaming Configuration Data

To provide configuration data as a data stream, a client application emits the **<config>** or **<config-text>** tag elements within the **<rpc>** and **<edit-config>** tag elements. To specify the configuration elements to change, the application emits Junos XML or CLI configuration statements representing all levels of the configuration hierarchy from the root (represented by the **<configuration>** or **<configuration-text>** tag element) down to each element to change. The Junos XML notation is the same as that used to request configuration information.

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>
        <config>
            <configuration>
                <!-- configuration changes -->
            </configuration>
        </config>
    </edit-config>
</rpc>
]]>]]>
```

```
<rpc>
    <edit-config>
        <target>
            <candidate/>
        </target>
        <config-text>
            <configuration-text>
                <!-- configuration changes -->
            </configuration-text>
        </config-text>
    </edit-config>
</rpc>
```

```
]]>]]>
```

For more detailed information about the mappings between Junos OS configuration elements and Junos XML tag elements, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 19. The CLI configuration statement notation is further described in the *CLI User Guide*.

The following example shows how to provide Junos XML configuration data in a data stream to configure the **messages** system log file:

Client Application **NETCONF Server**

```
<rpc message-id="messageID">
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <system>
          <syslog>
            <file>
              <name>messages</name>
              <contents>
                <name>any</name>
                <warning/>
              </contents>
              <contents>
                <name>authorization</name>
                <info/>
              </contents>
            </file>
          </syslog>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

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

T2135

Formatting Data: Junos XML versus CLI Configuration Statements

The NETCONF **<copy-config>** and **<edit-config>** operations accept one of two formats for Junos OS configuration data: Junos XML or CLI configuration statements. The choice between one data format over the other is personal preference.

NOTE: When managing devices running Junos OS, a client application can use the Junos XML protocol <load-configuration> operation in a NETCONF session to upload configuration data formatted using JSON or configuration mode set commands, in addition to Junos XML or CLI configuration statement formats.

If you are supplying the configuration changes in the form of data files, you enclose the data filename and path within **<url>** tags. By default, these tags specify that the referenced data files contain Junos XML-formatted configuration data. Thus, the following code declares that the data within the file is Junos XML elements:

```
<url>dataFile</url>
```

To specify that the data file contains CLI configuration statements, , include the **format="text"** attribute in the **<url>** tag.

```
<url format=”text”>dataFile</url>
```

When streaming data, you specify the data format by selecting one of two tags: **<config>** for Junos XML elements and **<config-text>** for CLI configuration statements.

In the following example, the **<configuration>** element encloses Junos XML-formatted configuration data:

```
<config>
  <configuration>
    <system>
      <services>
        <ssh>
           <protocol-version>v2</protocol-version>
        </ssh>
      </services>
    </system>
  </configuration>
</config>
```

In the following example, the **<configuration-text>** element encloses the same data formatted as CLI configuration statements:

```
<config-text>
  <configuration-text>
    system {
      services {
        ssh {
           protocol-version v2 ;
        }
      }
    }
  </configuration-text>
</config-text>
```

Related Documentation

- Editing the Configuration Using NETCONF on page 139
- <copy-config> on page 80
- <edit-config> on page 82

Setting the Edit Configuration Mode in a NETCONF Session

When sending configuration data to the NETCONF server, you can specify how the device should handle the configuration changes. This is known as the edit configuration mode. You can set the edit configuration mode globally for the entire session. You can also set the edit mode for only specific elements within the session.

Devices running Junos OS have the following edit configuration modes:

- **merge**—The device merges new configuration data into the existing configuration data. This is the default.
- **replace**—The device replaces existing configuration data with the new configuration data.
- **none**—The device does not change the existing configuration unless the new configuration element includes an operation attribute.

To set the edit configuration mode globally for the session, include the **<default-operation>** element with the desired mode as a child element of **<edit-config>**.

```
<rpc>
  <edit-config>
    <default-operation>mode</default-operation>
  <edit-config>
</rpc>
```

To specify the edit configuration mode for an individual element, include the **operation** attribute and desired mode in that element's tag.

```
<rpc>
  <edit-config>
    <config>
      <configuration>
        <protocols>
          <rip>
            <message-size operation="replace">255</message-size>
          </rip>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
```

You can also set a global edit configuration mode for an entire set of configuration changes and specify a different mode for individual elements that you want handled in a different manner. For example:

```
<rpc>
  <edit-config>
```

```
    <default-operation>merge</default-operation>
    <config>
      <configuration>
        <protocols>
          <rip>
            <message-size operation="replace">255</message-size>
          </rip>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
```

The edit configuration modes are discussed in more detail in the following sections:

- Specifying the merge Data Mode on page 148
- Specifying the replace Data Mode on page 149
- Specifying the none (no-change) Data Mode on page 149

Specifying the merge Data Mode

By default, the NETCONF server *merges* new configuration data into the candidate configuration or open configuration database. Thus, if you do not specify an edit configuration mode, the device merges the new configuration elements into the existing configuration.

Merging configurations is performed according to the following rules. (The rules also apply when updating configuration data in an open configuration database, for example, the ephemeral 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 configuration but not in the new configuration remains unchanged.
- A configuration element that exists in the new configuration but not in the candidate configuration is added to the candidate configuration.
- If a configuration element exists in both configurations, the following results occur:
  - If a child statement of the configuration element (represented by a child tag element) exists in the candidate configuration but not in the new configuration, it remains unchanged.
  - If a child statement exists in the new configuration but not in the candidate, it is added to the candidate configuration.
  - If a child statement exists in both configurations, the value in the new data replaces the value in the candidate configuration.

To explicitly specify that data be merged, the application includes the **<default-operation>** tag element with the value **merge** in the **<edit-config>** tag element.

```
<rpc>
    <edit-config>
```

```
        <default-operation>merge</default-operation>
        <!-- other child tag elements of the <edit-config> tag element -->
    </edit-config>
</rpc>
]]>]]>
```

Specifying the replace Data Mode

In the *replace* edit configuration mode, the new configuration data completely replaces the data in the candidate configuration or open configuration database. To specify that the data be replaced, the application includes the **<default-operation>** tag element with the value **replace** in the **<edit-config>** tag element.

```
<rpc>
    <edit-config>
        <default-operation>replace</default-operation>
    </edit-config>
</rpc>
]]>]]>
```

We recommend using the global replace mode only when you plan to completely overwrite the existing configuration with new configuration data. Furthermore, when the edit configuration mode is set to **replace**, we do not recommend using the **operation** attribute for individual configuration elements.

You can also replace individual configuration elements while merging or creating others. See "Replacing Configuration Elements Using NETCONF" on page 167.

Specifying the none (no-change) Data Mode

In the **none** (*no-change*) edit configuration mode, changes to the configuration are ignored. This mode is useful when you are deleting elements, and it prevents the NETCONF server from creating parent hierarchy levels for an element that is being deleted. For more information, see "Deleting Configuration Elements Using NETCONF" on page 161.

To set the no-change edit configuration mode globally, the application includes the **<default-operation>** tag element with the value **none** in the **<edit-config>** tag element.

```
<rpc>
  <edit-config>
    <default-operation>none</default-operation>
  </edit-config>
</rpc>
```

NOTE: If the new configuration data includes a configuration element that is not in the existing configuration, the NETCONF server returns an error. We recommend using mode none only when removing configuration elements from the configuration. When creating or modifying elements, applications must use merge mode.

When you use the **<default-operation>** tag to globally set the edit configuration mode to **none** to indicate the no-change mode, you can still override this mode and specify a different edit configuration mode for individual elements by including the **operation** attribute in the element's tag. For example:

```
<rpc>
  <edit-config>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <system>
          <services>
            <outbound-ssh>
              <client>
                <name>test</name>
                <device-id>test</device-id>
                <keep-alive>
                  <retry operation="merge">4</retry>
                  <timeout operation="merge">15</timeout>
                </keep-alive>
              </client>
            </outbound-ssh>
          </services>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
```

Related Documentation

- Deleting Configuration Elements Using NETCONF on page 161

Handling Errors While Editing the Candidate Configuration in a NETCONF Session

In a NETCONF session with a device running Junos OS, you can use NETCONF XML management protocol operations along with Junos XML or command-line interface (CLI) configuration statements to change the configuration on a routing, switching, or security platform. If the NETCONF server cannot incorporate the configuration data, the server returns the **<rpc-error>** tag element with information explaining the reason for the failure. By default, when the NETCONF server encounters an error while incorporating new configuration data into the candidate configuration, it halts the incorporation process. You can explicitly specify that the NETCONF server ignore errors or halt on error when incorporating new configuration data by including the **<error-option>** tag element.

A client application can explicitly specify that the NETCONF server stop incorporating new configuration data when it encounters an error. The application includes the **<error-option>** tag element with the value **stop-on-error** in the **<edit-config>** tag element.

```
<rpc>
    <edit-config>
        <error-option>stop-on-error</error-option>
        <!-- other child tag elements of the <edit-config> tag element -->
    </edit-config>
</rpc>
```

```
]]>]]>
```

Alternatively, the application can specify that the NETCONF server continue to incorporate new configuration data when it encounters an error. The application includes the **<error-option>** tag element with the value **ignore-error** in the **<edit-config>** tag element.

```
<rpc>
    <edit-config>
        <error-option>ignore-error</error-option>
        <!-- other child tag elements of the <edit-config> tag element -->
    </edit-config>
</rpc>
]]>]]>
```

The client application can include the optional **<test-option>** tag element described in the NETCONF specification. Regardless of the value provided, the NETCONF server for the Junos OS performs a basic syntax check on the configuration data in the **<edit-config>** tag element. When the **<test-option>** tag is included, NETCONF performs a complete syntactic and semantic validation in response to the **<commit>** and **<validate>** tag elements (that is, when the configuration is committed or explicitly checked), but not in response to the **<edit-config>** tag element.

Related Documentation

- Editing the Configuration Using NETCONF on page 139
- Verifying the Candidate Configuration Syntax Using NETCONF on page 173
- Committing the Candidate Configuration Using NETCONF on page 174
- Uploading and Formatting Configuration Data in a NETCONF Session on page 141

Replacing the Candidate Configuration Using NETCONF

In a NETCONF session with a device running Junos OS, a client application can replace the entire candidate configuration or all data in the open configuration database, either with new data or by rolling back to a previous configuration or a rescue configuration.

NOTE: If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before loading the configuration data, Junos OS performs the requested operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

The following sections discuss how to replace configuration data in the candidate configuration or open configuration database. The client application must commit the configuration after replacing the data to make it the active configuration on the device.

- Using <copy-config> to Replace the Configuration on page 152
- Using <edit-config> to Replace the Configuration on page 152

- Rolling Back to a Previously Committed Configuration on page 153
- Replacing the Candidate Configuration with the Rescue Configuration on page 154

Using <copy-config> to Replace the Configuration

One method for replacing the entire candidate configuration or all data in the open configuration database is to use the **<copy-config>** operation. The **<target>** tag encloses the **<candidate/>** tag to indicate that the new configuration data replaces either the data in the open configuration database (if the client application issued the Junos XML protocol **<open-configuration>** operation prior to executing the **<copy-config>** operation), or if there is no open database, the data in the candidate configuration.

The **<source>** element encloses the **<url>** element, which specifies the filename that contains the new configuration data. When the configuration data is formatted as Junos XML tag elements, set the **<url> format** attribute to **xml** or omit the attribute. When the configuration data is formatted as CLI configuration statements, set the **<url>** format attribute to **text**.

```
<rpc>
    <copy-config>
        <target>
            <candidate/>
        </target>
        <source>
            <url format="(xml | text)">
                <!-- location specifier for file containing the new
configuration -->
            </url>
        </source>
    </copy-config>
</rpc>
]]>]]>
```

Using <edit-config> to Replace the Configuration

Another method for replacing the entire candidate configuration or all data in the open configuration database is to use the **<edit-config>** operation and set the edit configuration mode to **replace** as a global variable. The application includes the **<default-operation>** tag element with the value **replace** in the **<edit-config>** tag element, as described in "Setting the Edit Configuration Mode in a NETCONF Session" on page 147. The **<target>** tag encloses the **<candidate/>** tag to indicate that the new configuration data replaces either the data in the open configuration database (if the client application issued the Junos XML protocol **<open-configuration>** operation prior to executing the **<edit-config>** operation), or if there is no open database, the data in the candidate configuration.

To specify the new configuration data, the application includes a **<config>** or **<config-text>** tag element that contains the data, or it includes a **<url>** tag element that names the file containing the data as discussed in "Uploading and Formatting Configuration Data in a NETCONF Session" on page 141.

```
<rpc>
    <edit-config>
```

```
        <target>
            <candidate/>
        </target>
        <default-operation>replace</default-operation>

    <!-- EITHER -->
        <config>
            <configuration>
                <!-- Junos XML configuration data -->
            </configuration>
        </config>
    <!-- OR -->
        <config-text>
            <configuration-text>
                <!-- configuration data in text format  -->
            </configuration-text>
        </config-text>
    <!-- OR -->
        <url>
            <!-- location specifier for file containing changes -->
        </url>

    </edit-config>
</rpc>
]]>]]>
```

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.

Starting in Junos OS Release 18.1R1, a NETCONF application can execute the **<rollback-config>** RPC to replace either the candidate configuration or all data in the open configuration database with a previously committed configuration. To roll back the configuration, the application emits the **<rollback-config>** element with the **<index>** child element, which specifies 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: NETCONF applications can also use the Junos XML protocol <load-configuration> operation with the rollback attribute to roll back the configuration.

For example, to load the configuration with a rollback index of 1, the client application emits the following RPC:

```
<rpc>
  <rollback-config>
```

```
      <index>1</index>
    </rollback-config>
  </rpc>
  ]]>]]>
```

The NETCONF server indicates that the load operation was successful by returning the **<rollback-config-results>** and **<ok/>** elements in its RPC reply.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/18.1R1/junos">
  <rollback-config-results>
    <ok/>
  </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 **<rpc-error>** element with information about the error.

Replacing the Candidate Configuration with the Rescue Configuration

A rescue configuration allows you to define a known working configuration or a configuration with a known state that you can restore at any time. 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.

Starting in Junos OS Release 18.1R1, a NETCONF application can execute the **<rollback-config>** RPC to replace either the candidate configuration or all data in the open configuration database with the device's rescue configuration. To load the rescue configuration, the application emits the **<rollback-config>** element and **<rescue/>** child tag. The rescue configuration must exist on the device before you can load it.

NOTE: NETCONF applications can also use the Junos XML protocol **<load-configuration>** operation with the **rescue** attribute to load the rescue configuration.

For example, to load the rescue configuration, the client application emits the following RPC:

```
<rpc>
  <rollback-config>
    <rescue/>
  </rollback-config>
```

```
  </rpc>
  ]]>]]>
```

The NETCONF server indicates that the load operation was successful by returning the **<rollback-config-results>** and **<ok/>** elements in its RPC reply.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/18.1R1/junos">
  <rollback-config-results>
    <ok/>
  </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 **<rpc-error>** element with information about the error.

Related Documentation

- Setting the Edit Configuration Mode in a NETCONF Session on page 147
- Replacing Configuration Elements Using NETCONF on page 167
- Uploading and Formatting Configuration Data in a NETCONF Session on page 141
- <copy-config> on page 80
- <edit-config> on page 82

Rolling Back Uncommitted Changes in the Candidate Configuration Using NETCONF

In a NETCONF session with a device running Junos OS, the client application can roll back the candidate configuration to the current running configuration, which removes any uncommitted changes from the candidate configuration. This operation is equivalent to the CLI configuration mode **rollback 0** command.

To roll back the candidate configuration to the current running configuration, enclose the **<discard-changes>** tag within the **<rpc>** element.

```
  <rpc>
      <discard-changes/>
  </rpc>
  ]]>]]>
```

After you issue the **</discard-changes>** tag, the NETCONF server indicates that it successfully discarded the changes by returning the **<ok/>** tag.

Related Documentation

- Replacing the Candidate Configuration Using NETCONF on page 151
- Retrieving a Previous (Rollback) Configuration Using NETCONF on page 233
- <discard-changes/> on page 82

Deleting the Configuration Using NETCONF

In a NETCONF session with a device running Junos OS, the **<delete-config>** tag element enables you to delete all configuration data in the current candidate configuration or in the open configuration database. Exercise caution when issuing the **<delete-config>** tag element. If you commit an empty candidate configuration, the device will go offline.

To delete the candidate configuration or all data in the open configuration database, insert the **<delete-config>** tag element in the **<rpc>** element. The **<target>** tag encloses the **<candidate/>** tag, which can refer to either the candidate configuration or the open configuration database. If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing a **<delete-config>** operation, Junos OS performs the operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

```
<rpc>
  <delete-config>
    <target>
      <candidate/>
    </target>
  </delete-config>
</rpc>
```

WARNING: If you take the device offline, you will need to access the device through the console port on the device. From this console, you can access the CLI and perform a rollback to a suitable configuration. For more information about the console port, see the hardware manual for your specific device.

Related Documentation

- Deleting Configuration Elements Using NETCONF on page 161
- Replacing the Candidate Configuration Using NETCONF on page 151
- Rolling Back Uncommitted Changes in the Candidate Configuration Using NETCONF on page 155
- <delete-config> on page 81

Changing Individual Configuration Elements Using NETCONF

In a NETCONF session with a device running Junos OS, a client application can change individual configuration elements in the existing configuration by using the **<edit-config>** tag element. By default, the NETCONF server merges new configuration data into the existing configuration. However, a client application can also replace, create, or delete individual configuration elements (hierarchy levels or configuration objects). The same basic tag elements are emitted for all operations: **<config>**, **<config-text>**, or **<url>** tag sub-elements within the **<edit-config>** tag element.

Within the **<edit-config>** element, the **<target>** element encloses the **<candidate/>** tag, which can refer to either the candidate configuration or the open configuration database. If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing the **<edit-config>** operation, Junos OS performs the operation on the open configuration database. Otherwise, the operation is performed on the candidate configuration.

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>


  <!-- EITHER -->
    <config>
      <configuration>
        <!-- tag elements representing the configuration elements to change -->
      </configuration>
    </config>
  <!-- OR -->
    <config-text>
      <configuration-text>
        <!-- configuration data in text format -->
      </configuration-text>
    </config-text>
  <!-- OR -->
    <url>
      <!-- location specifier for file containing changes -->
    </url>


  </edit-config>
</rpc>
]]>]]>
```

The application includes the configuration data within the **<config>** or **<config-text>** tag elements or in the file specified by the **<url>** tag element. To define a configuration element, the application includes the tag elements representing all levels of the configuration hierarchy from the root down to the immediate parent level for the element. To represent the element, the application includes its container tag element. The child tags included within the container element depend on the operation.

For more information about the tag elements that represent configuration statements, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 19. For information about the tag elements for a specific configuration element, see the *Junos XML API Configuration Developer Reference*.

The NETCONF server indicates that it changed the configuration in the requested way by enclosing the **<ok/>** tag in the **<rpc-reply>** tag element:

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



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

Related Documentation

- Creating Configuration Elements Using NETCONF on page 159
- Deleting Configuration Elements Using NETCONF on page 161
- Merging Configuration Elements Using NETCONF on page 158
- Replacing Configuration Elements Using NETCONF on page 167

Merging Configuration Elements Using NETCONF

In a NETCONF session with a device running Junos OS, to merge configuration elements, including hierarchy levels or configuration objects, into the existing configuration in the candidate configuration or the open configuration database (if the client application issued the Junos XML protocol **<open-configuration>** operation prior to executing the **<edit-config>** operation), a client application emits the basic tag elements described in "Changing Individual Configuration Elements Using NETCONF" on page 156.

To represent each element to merge in (either within the **<config>** or **<config-text>** tag elements or in the file specified by the **<url>** tag element), the application includes the tag elements representing its parent hierarchy levels and its container tag element, as described in "Changing Individual Configuration Elements Using NETCONF" on page 156. Within the container tag, the application includes each of the element's identifier tag elements (if it has them) and the tag element for each child to add or for which to set a different value. In the following, the identifier tag element is called **<name>**:

```
<configuration>
    <!-- opening tags for each parent of the element -->
        <element>
            <name>identifier</name>
            <!-- - child tag elements to add or change -->
        </element>
    <!-- closing tags for each parent of the element -->
</configuration>
```

The NETCONF server merges in the new configuration element according to the rules specified in "Setting the Edit Configuration Mode in a NETCONF Session" on page 147. As described in that section, the application can explicitly specify merge mode by including the **<default-operation>** tag element with the value **merge** in the **<edit-config>** tag element.

The following example shows how to merge information for a new interface called so-3/0/0 into the **[edit interfaces]** hierarchy level in the candidate configuration:

Client Application **NETCONF Server**

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <interfaces>
          <interface>
            <name>so-3/0/0</name>
            <unit>
              <name>0</name>
              <family>
                <inet>
                  <address>
                    <name>10.0.0.1/8</name>
                  <address>
                </inet>
              </family>
            </unit>
          </interface>
        </interfaces>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

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

T2120

Related Documentation

- Changing Individual Configuration Elements Using NETCONF on page 156
- Creating Configuration Elements Using NETCONF on page 159
- Deleting Configuration Elements Using NETCONF on page 161
- Replacing Configuration Elements Using NETCONF on page 167
- Setting the Edit Configuration Mode in a NETCONF Session on page 147

Creating Configuration Elements Using NETCONF

In a NETCONF session with a device running Junos OS, to create configuration elements, including hierarchy levels or configuration objects, that do not already exist in the target configuration, which can be either the candidate configuration or the open configuration database (if the client application issued the Junos XML protocol **<open-configuration>** operation prior to executing the **<edit-config>** operation), a client application emits the basic tag elements described in "Changing Individual Configuration Elements Using NETCONF" on page 156.

To represent each configuration element being created (either within the **<config>** or **<config-text>** tag elements or in the file specified by the **<url>** tag element), the

application emits the tag elements representing its parent hierarchy levels and its container tag element, as described in "Changing Individual Configuration Elements Using NETCONF" on page 156. Within the container tag, the application includes each of the element's identifier tag elements (if it has them) and all child tag elements (with values, if appropriate) that are being defined for the element. In the following, the identifier tag element is called **<name>**. The application includes the **operation="create"** attribute in the opening container tag:

```
<configuration>
    <!-- opening tags for each parent of the element -->
        <element operation="create">
            <name>identifier</name>  <!-- if element has an identifier -->
            <!-- other child tag elements -->
        </element>
    <!-- closing tags for each parent of the element -->
</configuration>
```

The NETCONF server adds the new element to the target configuration only if there is no existing element with that name (for a hierarchy level) or with the same identifiers (for a configuration object).

The following example shows how to enable OSPF on a device if it is not already configured:

Client Application

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <protocols>
          <ospf operation="create">
            <area>
              <name>0</name>
              <interface>
                <name>at-0/1/0.100</name>
              </interface>
            </area>
          </ospf>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

NETCONF Server

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

T2122

Related Documentation

- Changing Individual Configuration Elements Using NETCONF on page 156
- Deleting Configuration Elements Using NETCONF on page 161

- Merging Configuration Elements Using NETCONF on page 158
- Replacing Configuration Elements Using NETCONF on page 167
- Setting the Edit Configuration Mode in a NETCONF Session on page 147

Deleting Configuration Elements Using NETCONF

In a NETCONF session with a device running Junos OS, to delete a configuration element, including hierarchy levels or configuration objects, from the existing configuration in the candidate configuration or the open configuration database (if the client application issued the Junos XML protocol **<open-configuration>** operation prior to executing the **<edit-config>** operation), a client application emits the basic tag elements described in "Changing Individual Configuration Elements Using NETCONF" on page 156. It also emits the **<default-operation>** tag element with the value **none** to change the default mode to no-change.

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>none</default-operation>

  <!-- EITHER -->
    <config>
      <configuration>
        <!-- tag elements representing the configuration elements to delete -->
      </configuration>
    </config>
  <!-- OR -->
    <url>
      <!-- location specifier for file containing elements to delete -->
    </url>

  </edit-config>
</rpc>
]]>]]>
```

In no-change mode, existing configuration elements remain unchanged unless the corresponding element in the new configuration has the **operation="delete"** attribute in its opening tag. This mode prevents the NETCONF server from creating parent hierarchy levels for an element that is being deleted. We recommend that the only operation performed in no-change mode be deletion. When merging, replacing, or creating configuration elements, client applications use merge mode.

To represent each configuration element being deleted (either within the **<config>** tag element or in the file named by the **<url>** tag element), the application emits the tag elements representing its parent hierarchy levels, as described in "Changing Individual Configuration Elements Using NETCONF" on page 156. The tag element in which the

operation="delete" attribute is included depends on the element type, as described in the following sections:

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 **operation="delete"** attribute in the empty tag that represents the level:

```
<configuration>
    <!-- opening tags for each parent level -->
        <level-to-delete operation="delete"/>
    <!-- closing tags for each parent level -->
</configuration>
```

We recommend that the application set the default mode to no-change by including the **<default-operation>** tag element with the value **none**, as described in "Setting the Edit Configuration Mode in a NETCONF Session" on page 147. For more information about hierarchy levels and container objects, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 19.

The following example shows how to remove the **[edit protocols ospf]** hierarchy level of the candidate configuration:

Client Application

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <protocols>
          <ospf operation="delete"/>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

NETCONF Server

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

T2123

Deleting a Configuration Object That Has an Identifier

To delete a configuration object that has an identifier, a client application includes the **operation="delete"** attribute in the container tag element for the object. Inside the container tag element, it includes the identifier tag element only, not any tag elements that represent other characteristics. In the following, the identifier tag element is called **<name>**:

```
<configuration>
    <!-- opening tags for each parent of the object -->
        <object operation="delete">
            <name>identifier</name>
        </object>
    <!-- closing tags 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.

We recommend that the application set the default mode to no-change by including the **<default-operation>** tag element with the value **none**, as described in "Setting the Edit Configuration Mode in a NETCONF Session" on page 147. For more information about identifiers, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 19.

The following example shows how to remove the user object **barbara** from the **[edit system login user]** hierarchy level in the candidate configuration:

Client Application **NETCONF Server**

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <system>
          <login>
            <user operation="delete">
              <name>barbara</name>
            </user>
          </login>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

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

T2124

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 **operation="delete"** attribute in the tag element for the option. In the following, the identifier tag element for the object is called **<name>**. (For information about deleting an option that can take multiple values, see "Deleting Values from a Multi-value Option of a Configuration Object" on page 165.)

```
<configuration>
    <!-- opening tags for each parent of the object -->
        <object>
            <name>identifier</name>  <!-- if object has an identifier -->
            <option1 operation="delete">
            <option2 operation="delete">
            <!-- tag elements for other options to delete -->
        </object>
    <!-- closing tags for each parent of the object -->
</configuration>
```

We recommend that the application set the default mode to no-change by including the **<default-operation>** tag element with the value **none**, as described in "Setting the Edit Configuration Mode in a NETCONF Session" on page 147. For more information about options, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 19.

The following example shows how to remove the fixed-form **disable** option at the **[edit forwarding-options sampling]** hierarchy level:

Client Application **NETCONF Server**

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <forwarding-options>
          <sampling>
            <disable operation="delete"/>
          </sampling>
        </forwarding-options>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

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

T2125

Deleting Values from a Multi-value 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 **operation="delete"** attribute in the opening tag for each value. It does not include tag elements that represent values to be retained. The identifier tag element in the following is called **<name>**:

```
<configuration>
    <!-- opening tags for each parent of the parent object -->
        <parent-object>
            <name>identifier</name>
            <object operation="delete">value1</object>
```

```
            <object operation="delete">value2</object>
          </parent-object>
      <!-- closing tags for each parent of the parent object -->
  </configuration>
```

We recommend that the application set the default mode to no-change by including the **<default-operation>** tag element with the value **none**, as described in "Setting the Edit Configuration Mode in a NETCONF Session" on page 147. For more information about leaf statements with multiple values, see "Mapping Configuration Statements to Junos XML Tag Elements" on page 19.

The following example shows how to remove two of the permissions granted to the **user-accounts** login class:

Client Application

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>none</default-operation>
    <config>
      <configuration>
        <system>
          <login>
            <class>
              <name>user-accounts</name>
              <permissions operation="delete">configure</permissions>
              <permissions operation="delete">control</permissions>
            </class>
          </login>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

NETCONF Server

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

T2126

Related Documentation

- Changing Individual Configuration Elements Using NETCONF on page 156
- Deleting the Configuration Using NETCONF on page 156
- Creating Configuration Elements Using NETCONF on page 159
- Merging Configuration Elements Using NETCONF on page 158
- Replacing Configuration Elements Using NETCONF on page 167
- Setting the Edit Configuration Mode in a NETCONF Session on page 147

Replacing Configuration Elements Using NETCONF

In a NETCONF session with a device running Junos OS, to replace configuration elements, including hierarchy levels or configuration objects, in the candidate configuration, a client application emits the basic tag elements described in "Changing Individual Configuration Elements Using NETCONF" on page 156.

To represent the new definition for each configuration element being replaced (either within the **<config>** or **<config-text>** tag elements or in the file specified by the **<url>** tag element), the application emits the tag elements representing its parent hierarchy levels and its container tag element, as described in "Changing Individual Configuration Elements Using NETCONF" on page 156. Within the container tag, the application includes each of the element's identifier tag elements (if it has them) and all child tag elements (with values, if appropriate) that are being defined for the new version of the element. In the following example, the identifier tag element is called **<name>**. The application includes the **operation="replace"** attribute in the opening container tag:

```
<configuration>
    <!-- opening tags for each parent of the element -->
        <container-tag operation="replace">
            <name>identifier</name>
            <!-- other child tag elements -->
        </container-tag>
    <!-- closing tags for each parent of the element -->
</configuration>
```

The NETCONF server removes the existing element that has the specified identifiers and inserts the new element.

NOTE: The operation="replace" attribute is not supported when loading configuration data into the ephemeral configuration database.

The application can also replace all objects in the configuration in one operation. For instructions, see "Replacing the Candidate Configuration Using NETCONF" on page 151.

The following example shows how to grant new permissions for the object named **operator** at the **[edit system login class]** hierarchy level.

Client Application **NETCONF Server**

```
<rpc>
  <edit-config>
    <target>
      <candidate/>
    </target>
    <config>
      <configuration>
        <system>
          <login>
            <class operation="replace">
              <name>operator</name>
              <permissions>configure</permissions>
              <permissions>admin-control</permissions>
            </class>
          </login>
        </system>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

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

T2121

Related Documentation

- Changing Individual Configuration Elements Using NETCONF on page 156
- Creating Configuration Elements Using NETCONF on page 159
- Deleting Configuration Elements Using NETCONF on page 161
- Merging Configuration Elements Using NETCONF on page 158
- Setting the Edit Configuration Mode in a NETCONF Session on page 147

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*. 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 169
- Replacing Patterns Within a Hierarchy Level or Container Object That Has No Identifier on page 170
- Replacing Patterns for a Configuration Object That Has an Identifier on page 171

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 133
- *Using Global Replace in the Junos OS Configuration*
- *Common Regular Expressions to Use with the replace Command*
- *replace*

CHAPTER 11

Committing the Configuration Using NETCONF

Verifying the Candidate Configuration Syntax Using NETCONF

In a NETCONF session with a device running Junos OS, during the process of committing the candidate configuration or a private copy, the NETCONF server confirms that the configuration is syntactically correct. If the syntax check fails, the server does not commit the candidate 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.

In a NETCONF session with a device running Junos OS, to verify the syntax of the candidate configuration, a client application includes the **<validate>** and **<source>** tag elements and the **<candidate/>** tag in an **<rpc>** tag element:

```
<rpc>
    <validate>
        <source>
            <candidate/>
        </source>
    </validate>
</rpc>
]]>]]>
```

The NETCONF server confirms that the candidate configuration syntax is valid by returning the **<ok/>** tag in the **<rpc-reply>** tag element:

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

If the candidate configuration syntax is not valid, the server returns the **<rpc-reply>** element and **<rpc-error>** child element, which explains the reason for the error.

Related Documentation

- Committing the Candidate Configuration Using NETCONF on page 174
- Committing the Candidate Configuration Only After Confirmation Using NETCONF on page 175

Committing the Candidate Configuration Using NETCONF

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*.

In a NETCONF session with a device running Junos OS, to commit the candidate configuration, a client application encloses the **<commit/>** tag in an **<rpc>** tag element.

```
<rpc>
  <commit/>
</rpc>
]]>]]>
```

We recommend that the client application lock the candidate configuration before modifying it and emit the **<commit/>** 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.

The NETCONF server confirms that the commit operation was successful by returning the **<ok/>** tag in the **<rpc-reply>** tag element.

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

If the commit operation fails, the server returns the **<rpc-reply>** element and **<rpc-error>** child element, which explains the reason for the failure. The most common causes are semantic or syntactic errors in the candidate configuration.

In certain situations, if the commit operation succeeds but also returns a warning, the RPC reply includes both an **<rpc-error>** element with a severity level of warning and an **<ok/>** element. Starting in Junos OS Release 18.2R2, 18.3R2, and 18.4R1, when you configure the **rfc-compliant** statement at the **[edit system services netconf]** hierarchy level to enforce certain behaviors by the NETCONF server, the NETCONF server cannot return an RPC reply that includes both an **<rpc-error>** element and an **<ok/>** element. In this case, if the operation is successful but the server reply would include one or more **<rpc-error>** elements with a severity level of warning in addition to the **<ok/>** element, then the warnings are omitted.

Related Documentation

- Committing the Candidate Configuration Only After Confirmation Using NETCONF on page 175

- Locking and Unlocking the Candidate Configuration Using NETCONF on page 58

Committing the Candidate Configuration Only After Confirmation Using NETCONF

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 NETCONF 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>** and **<rpc>** tag elements:

```
<rpc>
    <commit>
        <confirmed/>
    </commit>
</rpc>
]]>]]>
```

To specify a different number of seconds for the rollback deadline, the application encloses a positive integer value in the **<confirm-timeout>** tag element:

```
<rpc>
    <commit>
        <confirmed/>
        <confirm-timeout>rollback-delay</confirm-timeout>
    </commit>
</rpc>
]]>]]>
```

NOTE: You cannot perform a confirmed commit operation on an instance of the ephemeral configuration database.

In either case, the NETCONF server confirms that it committed the candidate configuration temporarily by returning the **<ok/>** tag in the **<rpc-reply>**.

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

```
]]>]]>
```

If the NETCONF server cannot commit the candidate configuration, the **<rpc-reply>** element instead encloses an **<rpc-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 client application emits the **<confirmed/>** tag in a **<commit>** tag element again before the deadline passes. Optionally, it includes 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 600 seconds (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 the **<commit/>** tag enclosed in an **<rpc>** tag element before the rollback deadline passes. The rollback is canceled and the candidate configuration is committed immediately, as described in "Committing the Candidate Configuration Using NETCONF" on page 174. If the candidate configuration is still the same as the temporarily committed configuration, this effectively recommits the temporarily committed configuration.

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 NETCONF 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 a NETCONF Session" on page 60.

The following example shows how to commit the candidate configuration with a rollback deadline of 300 seconds.

Client Application

```
<rpc>
  <commit>
    <confirmed/>
    <confirm-timeout>300</confirm-timeout>
  </commit>
</rpc>
]]>]]>
```

NETCONF Server

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

Related Documentation

- Committing the Candidate Configuration Using NETCONF on page 174

CHAPTER 12

Using the Ephemeral Configuration Database

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

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 182.
- 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.

CAUTION: **We strongly recommend that you exercise caution when using the ephemeral configuration database, because committing invalid configuration data can corrupt the ephemeral database, which can cause Junos OS processes to restart or even crash 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 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.

NOTE: Starting in Junos OS Release 18.2R1, devices running Junos OS support configuring up to seven user-defined instances of the ephemeral configuration database. In earlier releases, you can configure up to eight user-defined instances.

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.

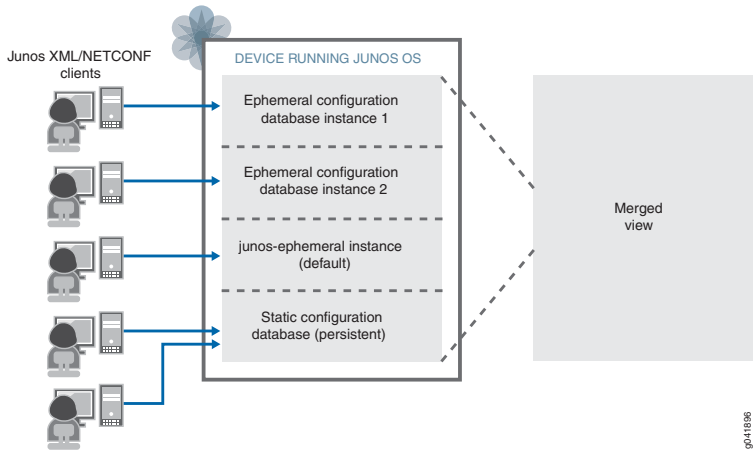
The Junos OS processes read the configuration data from both the static configuration database and the ephemeral configuration database. When one or more ephemeral database instances are in use and there is conflicting data, statements in a database with a higher priority override those in a database 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 180 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. 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.

CAUTION: 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. You must validate all configuration data before loading it into the ephemeral database and committing it on the device, because committing invalid configuration data can cause Junos OS processes to restart or even crash and result in disruption to the system or network.

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 179. 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 191
- *Committing and Synchronizing a Configuration on Redundant Control Planes Using the Junos XML Protocol*

Release History Table

Release	Description
18.2R1	Starting in Junos OS Release 18.2R1, devices running Junos OS support configuring up to seven user-defined instances of the ephemeral configuration database. In earlier releases, you can configure up to eight user-defined instances.

Related Documentation

- Enabling and Configuring Instances of the Ephemeral Configuration Database on page 184
- Example: Configuring the Ephemeral Configuration Database Using NETCONF on page 194

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 *interface-name* unit *logical-unit-number* alias *alias-name*]
[edit logical-systems]	[edit logical-systems *logical-system-name* interfaces *interface-name* unit *logical-unit-number* alias *alias-name*] [edit logical-systems *logical-system-name* policy-options prefix-list *name* apply-path *path*] [edit logical-systems *logical-system-name* protocols mstp] [edit logical-systems *logical-system-name* protocols rstp] [edit logical-systems *logical-system-name* protocols vstp] [edit logical-systems *logical-system-name* system processes routing]
[edit policy-options]	[edit policy-options prefix-list *name* apply-path *path*]
[edit protocols]	[edit protocols igmp] [edit protocols mld] [edit protocols mstp] [edit protocols rstp] [edit protocols vstp]
[edit routing-instances]	[edit routing-instances *instance-name* protocols mstp] [edit routing-instances *instance-name* protocols rstp] [edit routing-instances *instance-name* protocols vstp]
[edit security]	[edit security group-vpn member ipsec vpn] [edit security ssh-known-hosts host *hostname*]

NOTE: The ephemeral configuration database does not support configuring the **[edit security]** hierarchy on SRX Series Services Gateways.

[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 177

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 enable and configure multiple user-defined instances of the ephemeral configuration database.

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 Juniper Extension Toolkit Documentation.

Enabling Ephemeral Database Instances

The default ephemeral database instance is automatically enabled on devices running Junos OS that support configuring the ephemeral database. However, you must configure all user-defined instances of the ephemeral configuration database before using them. See Feature Explorer to verify the hardware platforms and software releases that support the ephemeral database.

To enable a user-defined instance of the ephemeral configuration database:

1. Configure the name of the instance, which must contain only alphanumeric characters, hyphens, and underscores and must not exceed 32 characters in length or use **default** as the name.

```
[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.

NOTE: Starting in Junos OS Release 17.1R3, 17.2R3, 17.3R3, 17.4R2, and 18.1R1, the name of an user-defined ephemeral database instance cannot be default.

2. Commit the configuration.

```
[edit system configuration-database ephemeral]
user@host# commit
```

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.

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
```

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.

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 certain load 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 | override | replace | set)" format="(text | json
    | xml)">
    <!--configuration-data-->
  </load-configuration>
</rpc>
```

NOTE: Starting in Junos OS Release 18.1R1, the ephemeral configuration database supports loading configuration data using the **<load-configuration> action** attribute values of **override** and **replace** in addition to the previously supported values of **merge** and **set**.

NOTE: The only acceptable format for **action="set"** is **"text"**. For more information about the **<load-configuration>** operation, see "<load-configuration>" on page 115.

```
<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.

Starting in Junos OS Release 18.2R1, the **show ephemeral-configuration** operational mode command uses a different syntax and supports filtering for displaying specific hierarchy levels. The new syntax is as follows:

- To view the configuration data in the default instance of the ephemeral configuration database, issue the **show ephemeral-configuration instance default** command.

  ```
  user@host> show ephemeral-configuration instance default
  ```

- To view the configuration data in a user-defined instance of the ephemeral configuration database, issue the **show ephemeral-configuration instance** ***instance-name*** command.

  ```
  user@host> show ephemeral-configuration instance 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 merge** command.

  ```
  user@host> show ephemeral-configuration merge
  ```

- To specify the scope of the configuration data to display, append the statement path of the requested hierarchy to the command. For example, the following command displays the configuration data at the **[edit system]** hierarchy level in the default instance of the ephemeral configuration database.

  ```
  user@host> show ephemeral-configuration instance default system
  ```

In Junos OS Release 18.1 and earlier releases:

- To view the configuration data in the default instance of the ephemeral configuration database, issue the **show ephemeral-configuration** command.

  ```
  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*** 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** command.

```
user@host> show ephemeral-configuration | display merge
```

Table 5 on page 191 outlines the **show ephemeral-configuration** commands for the various releases.

Table 5: show ephemeral-configuration Command

Action	18.2R1 and Later Releases	18.1 and Earlier Releases
View the configuration data in the default ephemeral instance	**show ephemeral-configuration instance default**	**show ephemeral-configuration**
View the configuration data in a user-defined ephemeral instance	**show ephemeral-configuration instance *instance-name***	**show ephemeral-configuration *instance-name***
View the complete post-inheritance configuration merged with the configuration data in all instances of the ephemeral database	**show ephemeral-configuration merge**	**show ephemeral-configuration | display merge**

Release History Table

Release	Description
18.2R1	Starting in Junos OS Release 18.2R1, the **show ephemeral-configuration** operational mode command uses a different syntax and supports filtering for displaying specific hierarchy levels.
18.1R1	Starting in Junos OS Release 18.1R1, the ephemeral configuration database supports loading configuration data using the **<load-configuration> action** attribute values of **override** and **replace** in addition to the previously supported values of **merge** and **set**.

Related Documentation

- Example: Configuring the Ephemeral Configuration Database Using NETCONF on page 194
- Understanding the Ephemeral Configuration Database on page 177
- Committing an Instance of the Ephemeral Configuration Database Using the NETCONF or Junos XML Protocol on page 191
- **ephemeral on page 410**
- **show ephemeral-configuration on page 432**

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.

CAUTION: 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. You must validate all configuration data before loading it into the ephemeral database and committing it on the device, because committing invalid configuration data can cause Junos OS processes to restart or even crash and result in disruption to the system or network.

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*.

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*
- Enabling and Configuring Instances of the Ephemeral Configuration Database on page 184
- Understanding the Ephemeral Configuration Database on page 177

Example: Configuring the Ephemeral Configuration Database Using NETCONF

The ephemeral database is an alternate configuration database that enables 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. This example shows how to enable an instance of the ephemeral configuration database and make updates to that instance in a NETCONF session.

Requirements

This example uses the following software components:

- MX Series router running Junos OS Release 16.2R2 or a later release.

Before you begin:

- Enable the NETCONF-over-SSH service on the device running Junos OS.

Overview

Multiple NETCONF and Junos XML protocol client applications can simultaneously load and commit configuration changes to a device running Junos OS by using ephemeral database instances. This example enables the ephemeral database instance **eph1** on a device running Junos OS and then configures the instance through a NETCONF session.

A client application must open an instance of the ephemeral configuration database in order to view or modify it. After establishing a NETCONF session, the client opens the ephemeral instance by using the Junos XML protocol **<open-configuration>** operation, which encloses the **<ephemeral-instance>** child tag and the name of the instance. Opening the ephemeral instance automatically acquires an exclusive lock on it.

The client then loads configuration data in text format into the **eph1** ephemeral instance. Because the configuration data is in text format, the **<load-configuration>** operation must include the **format="text"** attribute, and the configuration data must be enclosed in the **<configuration-text>** element.

This examples commits the configuration changes in the ephemeral instance by emitting the Junos XML protocol **<commit-configuration>** operation. The **<load-configuration> action="merge"** attribute only determines how the configuration data is merged into that instance of the ephemeral database. After you commit the changes to the ephemeral instance, Junos OS merges the configuration data into the active configuration according to the rules of prioritization. If there is conflicting data in the different configuration databases, statements in the **eph1** instance have a higher priority than those in the default ephemeral instance or the static configuration database. If there are other 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.

The **<close-configuration/>** operation closes the open ephemeral instance and releases the exclusive lock. The committed ephemeral data is retained until the device is rebooted, at which time the device deletes the configuration data in the **eph1** ephemeral instance as well as the data in all other ephemeral instances.

Configuration

Enabling the Ephemeral Database Instance

Step-by-Step Procedure

To enable the ephemeral database instance on the device running Junos OS:

1. Configure the name of the instance.

```
[edit]
user@host# set system configuration-database ephemeral instance eph1
```

2. Commit the configuration.

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

Results

From configuration mode, confirm your configuration by entering the **show system configuration-database** command. If the output does not display the intended configuration, repeat the instructions in this example to correct the configuration.

```
[edit]
user@host# show system configuration-database
ephemeral {
   instance eph1;
}
```

Configuring the Ephemeral Database Instance

Step-by-Step Procedure

To configure the ephemeral database instance and commit the changes from within a NETCONF session, a client application performs the following steps:

1. Open the ephemeral database instance.

```
<rpc>
  <open-configuration>
    <ephemeral-instance>eph1</ephemeral-instance>
  </open-configuration>
</rpc>
]]>]]>
```

2. Load the configuration data into the open ephemeral instance, and include the appropriate tags and attributes for that data.

```
<rpc>
  <load-configuration action="merge" format="text">
    <configuration-text>
      protocols {
        mpls {
          label-switched-path to-hastings {
```

```
            to 192.0.2.1;
          }
        }
      }
    </configuration-text>
  </load-configuration>
</rpc>
]]>]]>
```

3. If the **<load-configuration>** operation does not generate any errors, commit the configuration.

```
<rpc>
  <commit-configuration/>
</rpc>
]]>]]>
```

4. Close the ephemeral database instance.

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

Results

If there are no errors when opening or closing the database, the NETCONF server returns an empty **<rpc-reply>** element in response to the requests. The NETCONF server indicates a successful **<load-configuration>** operation by returning an empty **<ok/>** tag enclosed within the **<load-configuration-results>** and **<rpc-reply>** elements. Similarly, the NETCONF server indicates a successful **<commit-configuration>** operation by returning an empty **<ok/>** tag enclosed in an **<rpc-reply>** element.

Verification

Verifying the Commit

Purpose The NETCONF server's response to the commit operation should indicate the success or failure of the commit. You can also verify the success of the commit by reviewing the commit events for the ephemeral database in the system log file.

Action Review the system log file and display events that match **UI_EPHEMERAL**.

```
user@host> show log messages | match UI_EPHEMERAL
```

```
Feb 10 13:20:32  host mgd[5172]: UI_EPHEMERAL_COMMIT: User 'user' has requested
commit on 'eph1' ephemeral database
Feb 10 13:20:32  host mgd[5172]: UI_EPHEMERAL_COMMIT_COMPLETED: commit complete
on 'eph1' ephemeral database
```

Meaning The **UI_EPHEMERAL_COMMIT_COMPLETED** message tag indicates that the commit operation on the **eph1** instance was successful.

Verifying the Configuration Data in the Ephemeral Database Instance

Purpose Verify that the correct configuration data has been added to the ephemeral instance.

Action Within the NETCONF session, open the ephemeral database instance and retrieve the configuration.

```
<rpc>
  <open-configuration>
    <ephemeral-instance>eph1</ephemeral-instance>
  </open-configuration>
</rpc>
]]>]]>
<rpc>
  <get-configuration format="text"/>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/16.2R2/junos">
<configuration-text xmlns="http://xml.juniper.net/xnm/1.1/xnm">
## Last changed: 2017-02-10 13:20:32 PDT
protocols {
    mpls {
        label-switched-path to-hastings {
            to 192.0.2.1;
        }
    }
}
</configuration-text>
</rpc-reply>
]]>]]>
```

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

TIP: You can view the configuration data committed to an ephemeral database instance from the CLI by issuing the **show ephemeral-configuration instance *instance-name*** operational command in Junos OS Release 18.2R1 and later releases or by issuing the **show ephemeral-configuration *instance-name*** operational command in earlier releases.

Troubleshooting

Troubleshooting Issues When Opening the Ephemeral Instance

Problem You attempt to open an instance of the ephemeral database, and the server returns only an opening **<rpc-reply>** tag. For example:

```
<rpc>
     <open-configuration>
         <ephemeral-instance>eph1</ephemeral-instance>
     </open-configuration>
</rpc>
]]>]]>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/16.2R2/junos">
```

This issue can occur when another client has the exclusive lock on that instance.

Solution If another user has an exclusive lock on the ephemeral instance, a client application can issue remote procedure calls (RPCs) to update the ephemeral instance, but the operations on that ephemeral instance are not processed until the lock is released. When the lock is released, the server should issue the closing **</rpc-reply>** tag and process any RPCs emitted while the ephemeral instance was locked.

Alternatively, a client application can choose to update a different ephemeral instance, but with the caveat that different ephemeral instances have different priority levels when resolving conflicting configuration statements.

Troubleshooting Operational Issues

Problem The device running Junos OS does not execute operational changes that should occur as a result of committing certain configuration data to the ephemeral database instance, even though you have verified that the commit was successful and that the configuration data is present in the configuration for that ephemeral instance.

The operational changes might not occur if there is another user-defined ephemeral instance that has conflicting configuration data and a higher priority. If there is conflicting data in the ephemeral instances, statements in an instance with a higher priority override those in an instance 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.

Solution

You can verify the configured ephemeral instances and their priority order by issuing the **show configuration system configuration-database ephemeral** operational command on the device. Instances are listed in order from highest to lowest priority. If there are other instances that have a higher priority, review the configuration data in those instances to determine if there are conflicting statements. You can also display the merged view of the static and ephemeral configuration databases by issuing the **show ephemeral-configuration merge** command in Junos OS Release 18.2R1 and later releases or by issuing the **show ephemeral-configuration | display merge** command in earlier releases.

If your ephemeral instance has conflicting configuration data and a lower priority than another user-defined ephemeral instance, and the configuration at that hierarchy level should go into effect on the device, you must either delete the conflicting data in the other ephemeral instance or place your configuration data in a higher priority instance.

Related Documentation

- Understanding the Ephemeral Configuration Database on page 177
- Enabling and Configuring Instances of the Ephemeral Configuration Database on page 184
- **ephemeral on page 410**

PART 4

Requesting Operational and Configuration Information Using NETCONF

CHAPTER 13

Requesting Operational Information Using NETCONF

Requesting Operational Information Using NETCONF

Within a NETCONF 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 element corresponds to the **show interfaces** command, the **<get-chassis-inventory>** tag element requests the same information as the **show chassis hardware** command, and the **<get-system-inventory>** tag element 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 NETCONF 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 NETCONF Session" on page 205.

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 56.

If the client application requests the output in XML-tagged format, the NETCONF 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="URN" xmlns:junos="URL">
  <operational-response xmlns="URL-for-DTD">
    <!-- 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 NETCONF 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 NETCONF server encloses its response in an **<output>** tag element, which is enclosed in an **<rpc-reply>** tag.

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



```
      operational-response
    </output>
  </rpc-reply>
]]>]]>
```

If the client application requests the output in JSON format, the NETCONF server encloses the JSON data in the **<rpc-reply>** tag element.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  operational-response
</rpc-reply>
]]>]]>
```

Related Documentation

- Understanding the Request Procedure in a NETCONF Session on page 32
- Specifying the Output Format for Operational Information Requests in a NETCONF Session on page 205
- Requesting Configuration Data Using NETCONF on page 213

Specifying the Output Format for Operational Information Requests in a NETCONF Session

In a NETCONF 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 NETCONF server. Information can be returned as formatted ASCII text, JavaScript Object Notation (JSON), or XML-tagged format. 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 NETCONF 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 NETCONF 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="urn:ietf:params:xml:ns:netconf:base:1.0"
           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>
]]>]]>
```

JSON Format

Starting in Junos OS Release 14.2, you can display operational and configuration data in JSON format. To request that the NETCONF server return operational information in 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 NETCONF server formats the reply using JSON.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
           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>
]]>]]>
```

Starting in Junos OS Release 17.3R1, devices running Junos OS support emitting the device's operational state in compact JSON format, in which only objects that have multiple values are emitted as JSON arrays. To configure the device to emit compact JSON format, configure the **compact** statement at the **[edit system export-format state-data json]** hierarchy level.

ASCII Format

To request that the NETCONF 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 NETCONF 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="urn:ietf:params:xml:ns:netconf:base:1.0"
           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 NETCONF 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 NETCONF 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 NETCONF 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.

Release History Table

Release	Description
17.3R1	Starting in Junos OS Release 17.3R1, devices running Junos OS support emitting the device's operational state in compact JSON format, in which only objects that have multiple values are emitted as JSON arrays.
14.2	Starting in Junos OS Release 14.2, you can display operational and configuration data in JSON format.

CHAPTER 14

Requesting Configuration Information Using NETCONF

Requesting the Committed Configuration and Device State Using NETCONF

In a NETCONF session with a device running Junos OS, to request the most recently committed configuration and the device state information for a routing, switching, or security platform, a client application encloses the **<get>** tag in an **<rpc>** tag element. By including the **<filter>** tag element and appropriate child tag elements, the application can request specific portions of the configuration. If the **<filter>** element is omitted, the server returns the entire configuration. The optional **format** attribute specifies the return format for the configuration data.

```
<rpc>
  <get [format="(json | set | text | xml)"]>
    <filter type="subtree">
      <!-- tag elements representing the configuration elements to return -->
    </filter>
  </get>
</rpc>
]]>]]>
```

The **type="subtree"** attribute in the opening **<filter>** tag indicates that the client application is using Junos XML tag elements to represent the configuration elements about which it is requesting information.

The NETCONF server encloses its reply in the **<rpc-reply>** and **<data>** tag elements. Within the **<data>** element, the configuration data is enclosed in the **<configuration>**, **<configuration-text>**, **<configuration-set>**, or **<configuration-json>** element depending on the requested format, and the device information is enclosed in the **<database-status-information>** element. The server includes attributes in the opening **<configuration>** tag that indicate the XML namespace for the enclosed tag elements and when the configuration was last changed or committed. For example:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL" junos:changed-seconds="seconds"
      junos:changed-localtime="time">
      <!-- configuration data -->
    </configuration>
    <database-status-information>
      <database-status>
        <user>user</user>
        <terminal></terminal>
        <pid>pid</pid>
        <start-time junos:seconds="1416956595">2014–11–25 15:03:15 PST</start-time>
        <edit-path></edit-path>
      </database-status>
    </database-status-information>
  </data>
</rpc-reply>
]]>]]>
```

If there is no configuration data in the requested hierarchy, the RPC reply contains an empty **<configuration>** tag inside the **<data>** element unless the **rfc-compliant** statement is configured, in which case the **<configuration>** tag is omitted.

Related Documentation

- <get> on page 85
- Requesting Configuration Data Using NETCONF on page 213

Requesting Configuration Data Using NETCONF

In a NETCONF session with a device running Junos OS, to request configuration data for a routing, switching, or security platform, a client application encloses the **<get-config>**, **<source>**, and **<filter>** tag elements in an **<rpc>** tag element. By including the appropriate child tag element in the **<source>** tag element, the client application requests information from the active configuration or from the candidate configuration or open configuration database. By including the appropriate child tag elements in the **<filter>** tag element, the application can request the entire configuration or specific portions of the configuration.

```
<rpc>
  <get-config>
    <source>
      <!-- tag specifying the source configuration -->
      <( candidate | running )/>
    </source>
    <filter type="subtree">
      <!-- tag elements representing the configuration elements to return -->
    </filter>
  </get-config>
</rpc>
]]>]]>
```

The **type="subtree"** attribute in the opening **<filter>** tag indicates that the client application is using Junos XML tag elements to represent the configuration elements about which it is requesting information.

NOTE: If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing the **<get-config>** operation, setting the source to **<candidate/>** retrieves the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration.

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 Using NETCONF" on page 58.

The NETCONF server encloses its reply in **<rpc-reply>**, **<data>**, and **<configuration>** tag elements. It includes attributes in the opening **<configuration>** tag that indicate the XML namespace for the enclosed tag elements and when the configuration was last changed or committed. For information about the attributes of the **<configuration>** tag,

see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration attributes>
      <!-- JUNOS XML tag elements representing configuration elements -->
    </configuration>
  </data>
</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 character using the equivalent UTF-8 decimal character reference. For more information see "Understanding Character Encoding on Devices Running Junos OS" on page 56.

Client applications can also request other configuration-related information, including an XML schema representation of the configuration hierarchy or information about previously committed configurations.

Related Documentation

- Specifying the Source for Configuration Information Requests Using NETCONF on page 214
- Retrieving a Previous (Rollback) Configuration Using NETCONF on page 233
- Comparing Two Previous (Rollback) Configurations Using NETCONF on page 236
- Retrieving the Rescue Configuration Using NETCONF on page 238
- Specifying the Scope of Configuration Information to Return in a NETCONF Response on page 216
- Requesting an XML Schema for the Configuration Hierarchy Using NETCONF on page 241
- Requesting Operational Information Using NETCONF on page 203

Specifying the Source for Configuration Information Requests Using NETCONF

In a NETCONF session with a device running Junos OS, to request information from the candidate configuration or open configuration database, a client application includes the **<source>** element and **<candidate/>** tag within the **<rpc>** and **<get-config>** tag elements.

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <!-- tag elements representing the configuration elements to return -->
    </filter>
  </get-config>
</rpc>
]]>]]>
```

NOTE: If a client application issues the Junos XML protocol **<open-configuration>** operation to open a specific configuration database before executing the **<get-config>** operation, setting the source to **<candidate/>** retrieves the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration.

To request information from the active configuration—the one most recently committed on the device—a client application includes the **<source>** tag element and **<running/>** tag enclosed within the **<rpc>** and **<get-config>** tag elements.

```
<rpc>
  <get-config>
    <source>
      <running/>
    </source>
    <filter>
      <!-- tag elements representing the configuration elements to return -->
    </filter>
  </get-config>
</rpc>
]]>]]>
```

NOTE: If a client application is requesting the entire configuration, it omits the **<filter>** tag element.

The NETCONF server encloses its reply in **<rpc-reply>**, **<data>**, and **<configuration>** tag elements. In the opening **<configuration>** tag, it includes the **xmlns** attribute to specify the namespace for the enclosed tag elements.

When returning information from the candidate configuration or open configuration database, the NETCONF server includes attributes that indicate when the configuration last changed (they appear on multiple lines here only for legibility).

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

```
    <configuration xmlns="URL" junos:changed-seconds="seconds" \
      junos:changed-localtime="YYYY-MM-DD hh:mm:ss TZ">
       <!-- Junos XML tag elements representing the configuration -->
     </configuration>
  </data>
</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 NETCONF server includes attributes that indicate when the configuration was committed (they appear on multiple lines here only for legibility).

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL" junos:commit-seconds="seconds" \
      junos:commit-localtime="YYYY-MM-DD hh:mm:ss TZ" \
      junos:commit-user="username">
      <!-- Junos XML tag elements representing the configuration -->
    </configuration>
  </data>
</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.

Related Documentation

- Requesting Configuration Data Using NETCONF on page 213
- <get-config> on page 86

Specifying the Scope of Configuration Information to Return in a NETCONF Response

In a NETCONF session with a device running Junos OS, a client application can request the entire configuration or specific portions of the configuration by including the appropriate child tag elements in the **<filter>** tag element within the **<rpc>** and **<get-config>** tag elements.

```
<rpc>
  <get-config>
```

```
    <source>
      ( <candidate/> | <running/> )
    </source>
    <filter type="subtree">
      <!-- tag elements representing the configuration elements to return -->
    </filter>
  </get-config>
</rpc>
]]>]]>
```

The **type="subtree"** attribute in the opening **<filter>** tag indicates that the client application is using Junos XML tag elements to represent the configuration elements about which it is requesting information.

For information about requesting different amounts of configuration information, see the following topics:

- Requesting the Complete Configuration Using NETCONF on page 217
- Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using NETCONF on page 219
- Requesting All Configuration Objects of a Specified Type Using NETCONF on page 221
- Requesting Identifiers for Configuration Objects of a Specified Type Using NETCONF on page 223
- Requesting A Specific Configuration Object Using NETCONF on page 226
- Requesting Specific Child Tags for a Configuration Object Using NETCONF on page 228
- Requesting Multiple Configuration Elements Simultaneously Using NETCONF on page 232

Related Documentation

- Requesting Configuration Data Using NETCONF on page 213
- Specifying the Source for Configuration Information Requests Using NETCONF on page 214

Requesting the Complete Configuration Using NETCONF

In a NETCONF session with a device running Junos OS, to request the entire candidate configuration or the complete configuration in the open configuration database, a client application encloses **<get-config>** and **<source>** tag elements and the **<candidate/>** tag in an **<rpc>** tag element:

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
  </get-config>
</rpc>
```

```
]]>]]>
```

NOTE: If a client application issues the Junos XML protocol <open-configuration> operation to open a specific configuration database before executing the <get-config> operation, setting the source to <candidate/> retrieves the configuration data from the open configuration database. Otherwise, the server returns the configuration data from the candidate configuration.

To request the entire active configuration, a client application encloses **<get-config>** and **<source>** tag elements and the **<running/>** tag in an **<rpc>** tag element:

```
<rpc>
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>
]]>]]>
```

The NETCONF server encloses its reply in **<rpc-reply>**, **<data>**, and **<configuration>** tag elements. For information about the attributes in the opening **<configuration>** tag, see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration attributes>
      <!-- Junos XML tag elements representing the configuration -->
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

Related Documentation

- Requesting Configuration Data Using NETCONF on page 213
- Specifying the Source for Configuration Information Requests Using NETCONF on page 214
- Specifying the Scope of Configuration Information to Return in a NETCONF Response on page 216
- Retrieving the Rescue Configuration Using NETCONF on page 238
- Requesting an XML Schema for the Configuration Hierarchy Using NETCONF on page 241

Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using NETCONF

In a NETCONF 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 **<filter>** 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 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-config>
    <source>
      <!-- tag specifying the source configuration -->
    </source>
    <filter type="subtree">
      <configuration>
        <!-- opening tags for each parent of the requested level -->
          <level-or-container/>
        <!-- closing tags for each parent of the requested level -->
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

For information about the **<source>** tag element, see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

The NETCONF server returns the requested section of the configuration in **<data>** and **<rpc-reply>** tag elements. For information about the attributes in the opening **<configuration>** tag, see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration attributes>
      <!-- opening tags for each parent of the level -->
        <level-or-container>
          <!-- child tag elements of the level or container -->
        </level-or-container>
      <!-- closing tags for each parent of the level -->
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-config>** tag element.

For more information, see "Requesting Multiple Configuration Elements Simultaneously Using NETCONF" on page 232.

The following example shows how to request the contents of the **[edit system login]** hierarchy level in the candidate configuration.

Client Application

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <configuration>
        <system>
          <login/>
        </system>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL" \
        junos:changed-seconds="seconds" \
        junos:changed-localtime="timestamp">
      <system>
        <login>
          <user>
            <name>barbara</name>
            <full-name>Barbara Anderson</full-name>
            <class>superuser</class>
            <uid>632</uid>
          </user>
          <!- - other child tag elements of <login> - ->
        </login>
      </system>
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

T2128

Related Documentation

- Requesting Configuration Data Using NETCONF on page 213
- Specifying the Scope of Configuration Information to Return in a NETCONF Response on page 216
- Specifying the Source for Configuration Information Requests Using NETCONF on page 214
- Requesting Identifiers for Configuration Objects of a Specified Type Using NETCONF on page 223
- Requesting Multiple Configuration Elements Simultaneously Using NETCONF on page 232

Requesting All Configuration Objects of a Specified Type Using NETCONF

In a NETCONF session with a device running Junos OS, to request information about all configuration objects of a specified type in a hierarchy level, a client application emits a **<filter>** 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 immediate parent level for the object type. An empty tag returns all configuration objects of the requested object type and all child tags for each object. To return only specific child tags for the configuration objects, enclose the desired child tags in the opening and closing tags of the object. The entire request is enclosed in an **<rpc>** tag element:

```
<rpc>
  <get-config>
    <source>
      <!-- tag specifying the source configuration -->
    </source>
    <filter type="subtree">
      <configuration>
        <!-- opening tags for each parent of the requested object type -->
          <object-type>
            <!-- optionally select specific child tags -->
          </object-type>
        <!-- closing tags for each parent of the requested object type -->
       </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

For information about the **<source>** tag element, see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

This type of request is useful when the object's parent hierarchy level has more than one type of child object. If the requested object is the only child type that can occur in its parent hierarchy level, then this type of request yields the same output as a request for the complete parent hierarchy, which is described in "Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using NETCONF" on page 219.

The NETCONF server returns the requested objects in **<data>** and **<rpc-reply>** tag elements. For information about the attributes in the opening **<configuration>** tag, see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

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

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-config>** tag element. For more information, see "Requesting Multiple Configuration Elements Simultaneously Using NETCONF" on page 232.

The following example shows how to request complete information about all **radius-server** objects at the **[edit system]** hierarchy level in the candidate configuration.

Client Application

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <configuration>
        <system>
          <radius-server/>
        </system>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL" \
        junos:changed-seconds="seconds" \
        junos:changed-localtime="timestamp">
      <system>
        <radius-server>
          <name>10.25.34.166</name>
          <secret>$9$Pf3900REcr/9t...</secret>
          <timeout>5</timeout>
          <retry>3</retry>
        </radius-server>
        <radius-server>
          <name>10.25.6.204</name>
          <secret>$9$K5Kvxd2gJZUi-d...</secret>
          <timeout>5</timeout>
          <retry>3</retry>
        </radius-server>
      </system>
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

T2129

Related Documentation

- Requesting Configuration Data Using NETCONF on page 213
- Specifying the Source for Configuration Information Requests Using NETCONF on page 214
- Specifying the Scope of Configuration Information to Return in a NETCONF Response on page 216
- Requesting Identifiers for Configuration Objects of a Specified Type Using NETCONF on page 223

Requesting Identifiers for Configuration Objects of a Specified Type Using NETCONF

In a NETCONF 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 **<filter>** 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. The object type is represented by its container tag element enclosing an empty **<name/>** tag. (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.) The entire request is enclosed in an **<rpc>** tag element:

```
<rpc>
  <get-config>
    <source>
      <!-- tag specifying the source configuration -->
    </source>
    <filter type="subtree">
      <configuration>
        <!-- opening tags for each parent of the object type -->
          <object-type>
            <name/>
          </object-type>
        <!-- closing tags for each parent of the object type -->
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

For information about the **<source>** tag element, see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

NOTE: You cannot request only identifiers for object types that have multiple identifiers. However, for many such objects the identifiers are the only child tag elements, so requesting complete information yields the same output as requesting only identifiers. For instructions, see "Requesting All Configuration Objects of a Specified Type Using NETCONF" on page 221.

The NETCONF server returns the requested objects in **<data>** and **<rpc-reply>** tag elements (here, objects for which 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 Using NETCONF" on page 214.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration attributes>
      <!-- opening tags for each parent of the object type -->
        <first-object>
          <name>identifier-for-first-object</name>
        </first-object>
        <second-object>
          <name>identifier-for-second-object</name>
        </second-object>
        <!-- additional objects -->
      <!-- closing tags for each parent of the object type -->
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-config>** tag element. For more information, see "Requesting Multiple Configuration Elements Simultaneously Using NETCONF" on page 232.

The following example shows how to request the identifier for each BGP neighbor configured at the **[edit protocols bgp group next-door-neighbors]** hierarchy level in the candidate configuration.

Client Application **NETCONF Server**

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <configuration>
        <protocols>
          <bgp>
            <group>
              <name>next-door-neighbors</name>
              <neighbor>
                <name/>
              </neighbor>
            </group>
          </bgp>
        </protocols>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL" \
        junos:changed-seconds="seconds" \
        junos:changed-localtime="timestamp">
      <protocols>
        <bgp>
          <group>
            <name>next-door-neighbors</name>
            <neighbor>
              <name>10.2.35.188</name>
            </neighbor>
            <neighbor>
              <name>10.3.62.95</name>
            </neighbor>
            <neighbor>
              <name>10.4.122.9</name>
            </neighbor>
          </group>
        </bgp>
      </protocols>
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

T2130

Related Documentation

- Requesting Configuration Data Using NETCONF on page 213
- Specifying the Source for Configuration Information Requests Using NETCONF on page 214
- Specifying the Scope of Configuration Information to Return in a NETCONF Response on page 216
- Requesting All Configuration Objects of a Specified Type Using NETCONF on page 221

Requesting A Specific Configuration Object Using NETCONF

In a NETCONF session with a device running Junos OS, to request complete information about a specific configuration object, a client application emits a **<filter>** 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-config>
    <source>
      <!--tag specifying the source configuration -->
    </source>
    <filter type="subtree">
      <configuration>
        <!-- opening tags for each parent of the object -->
          <object>
            <name>identifier</name>
          </object>
        <!-- closing tags for each parent of the object -->
      </configuration>
    </filter >
  </get-config>
</rpc>
]]>]]>
```

For information about the **<source>** tag element, see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

The NETCONF server returns the requested object in **<data>** and **<rpc-reply>** tag elements (here, an object for which 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 Using NETCONF" on page 214.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration attributes>
      <!-- opening tags for each parent of the object -->
        <object>
          <name>identifier</name>
          <!-- other child tag elements of the object -->
        </object>
```



```
        <!-- closing tags for each parent of the object -->
      </configuration>
    </data>
  </rpc-reply>
  ]]>]]>
```

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-config>** tag element. For more information, see "Requesting Multiple Configuration Elements Simultaneously Using NETCONF" on page 232.

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 in the candidate configuration. To specify the desired object, the client application emits the **<name>local</name>** identifier tag element as the innermost tag element.

Client Application

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <configuration>
        <routing-options>
          <multicast>
            <scope>
              <name>local</name>
            </scope>
          </multicast>
        </routing-options>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL"  \
        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>
  </data>
</rpc-reply>
]]>]]>
```

T2131

Related Documentation

- Requesting Configuration Data Using NETCONF on page 213
- Specifying the Source for Configuration Information Requests Using NETCONF on page 214
- Specifying the Scope of Configuration Information to Return in a NETCONF Response on page 216
- Requesting Specific Child Tags for a Configuration Object Using NETCONF on page 228

Requesting Specific Child Tags for a Configuration Object Using NETCONF

In a NETCONF session with a device running Junos OS, to request specific child tag elements and descendents for configuration objects, a client application emits a **<filter>** 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 its container tag element. To request a specific configuration object, include the identifier tag element. 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. If you omit the identifier tag element, the server returns the child tags for all configuration objects of that type. To select specific child tags, the client application emits all desired child tag elements and descendents within the container tag element. The entire request is enclosed in an **<rpc>** tag element:

```
<rpc>
  <get-config>
    <source>
      <!-- tag specifying the source configuration -->
    </source>
    <filter type="subtree">
      <configuration>
        <!-- opening tags for each parent of the object -->
          <object>
            <name>identifier</name>
            <first-child/>
            <second-child/>
            <third-child>
              <!--tags for descendents-->
            </third-child>
            <!-- tag for each additional child to return -->
          </object>
        <!-- closing tags for each parent of the object -->
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

For information about the **<source>** tag element, see "Specifying the Source for Configuration Information Requests Using NETCONF" on page 214.

The NETCONF server returns the requested children of the object in **<data>** and **<rpc-reply>** tag elements (here, an object for which 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 Using NETCONF" on page 214.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration attributes>
      <!-- opening tags for each parent of the object -->
        <object>
          <name>identifier</name>
          <!-- requested child tags -->
        </object>
      <!-- closing tags for each parent of the object -->
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

The application can also request additional configuration elements of the same or other types by including the appropriate tag elements in the same **<get-config>** tag element. For more information, see "Requesting Multiple Configuration Elements Simultaneously Using NETCONF" on page 232.

The following example shows how to request only the address of the next-hop device for the 192.168.5.0/24 route at the **[edit routing-options static]** hierarchy level in the candidate configuration.

Client Application **NETCONF Server**

```
<rpc>
  <get-config>
    <source>
      <candidate/>
    </source>
    <filter>
      <configuration>
        <routing-options>
          <static>
            <route>
              <name>192.168.5.0/24</name>
              <next-hop/>
            </route>
          </static>
        </routing-options>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL" \
         junos:changed-seconds="seconds" \
         junos:changed-localtime="timestamp">
      <routing-options>
        <static>
          <route>
            <name>192.168.5.0/24</name>
            <next-hop>192.168.71.254</next-hop>
          </route>
        </static>
      </routing-options>
    </configuration>
  </data>
</rpc-reply>
]]>]]>
```

T2132

The following example shows how to request the addresses for all logical interfaces configured for each physical interface within the groups hierarchy level of the candidate configuration.

```
<rpc>
  <get-config>
      <source>
        <candidate/>
      </source>
      <filter type="subtree">
        <configuration>
          <groups>
            <interfaces>
              <interface>
                <unit>
                  <family>
                    <inet>
                      <address/>
                    </inet>
```

```
                    </family>
                  </unit>
                </interface>
              </interfaces>
            </groups>
          </configuration>
        </filter>
      </get-config>
    </rpc>
```

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <data>
    <configuration xmlns="URL" junos:commit-seconds=seconds
junos:commit-localtime="timestamp" junos:commit-user="user">
      <groups>
        <name>re0</name>
        <interfaces>
          <interface>
            <name>lo0</name>
            <unit>
              <name>0</name>
              <family>
                <inet>
                  <address>
                    <name>127.0.0.1/32</name>
                  </address>
                </inet>
              </family>
            </unit>
          </interface>
          <interface>
            <name>em0</name>
            <unit>
              <name>0</name>
              <family>
                <inet>
                  <address>
                    <name>198.51.100.1/24</name>
                  </address>
                  <address>
                    <name>198.51.100.11/24</name>
                  </address>
                </inet>
              </family>
            </unit>
          </interface>
        </interfaces>
      </groups>
    </configuration>
  </data>
</rpc-reply>
```

Related Documentation

- Requesting Configuration Data Using NETCONF on page 213
- Specifying the Source for Configuration Information Requests Using NETCONF on page 214

- Specifying the Scope of Configuration Information to Return in a NETCONF Response on page 216
- Requesting A Specific Configuration Object Using NETCONF on page 226
- Requesting Multiple Configuration Elements Simultaneously Using NETCONF on page 232

Requesting Multiple Configuration Elements Simultaneously Using NETCONF

In a NETCONF 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-config>** tag element. The request includes only one **<filter>** and **<configuration>** tag element (the NETCONF server returns an error if there is more than one of each).

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 <system> tag element -->
<rpc>
    <get-config>
        <source>
            <!-- tag specifying the source configuration -->
        </source>
        <filter type="subtree">
            <configuration>
                <system>
                    <services/>
                    <radius-server>
                        <name/>
                    </radius-server>
                </system>
            </configuration>
        </filter>
    </get-config>
</rpc>
]]>]]>
<!-- separate <system> tag element for each element -->
<rpc>
    <get-config>
        <source>
            <!-- tag specifying the source configuration -->
        </source>
        <filter type="subtree">
            <configuration>
                <system>
                    <services/>
                </system>
                <system>
                    <radius-server>
                        <name/>
                    </radius-server>
                </system>
```

```
            </configuration>
          </filter>
      </get-config>
  </rpc>
  ]]>]]>
```

The client can combine requests for any of the following types of information:

- Requesting a Configuration Hierarchy Level or Container Object Without an Identifier Using NETCONF on page 219
- Requesting All Configuration Objects of a Specified Type Using NETCONF on page 221
- Requesting Identifiers for Configuration Objects of a Specified Type Using NETCONF on page 223
- Requesting A Specific Configuration Object Using NETCONF on page 226
- Requesting Specific Child Tags for a Configuration Object Using NETCONF on page 228

Related Documentation

- Requesting Configuration Data Using NETCONF on page 213
- Specifying the Source for Configuration Information Requests Using NETCONF on page 214
- Specifying the Scope of Configuration Information to Return in a NETCONF Response on page 216

Retrieving a Previous (Rollback) Configuration Using NETCONF

In a NETCONF 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 NETCONF server encloses its response in **<rpc-reply>**, **<rollback-information>**, and **<configuration>** tag elements. The **<ok/>** 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 Using NETCONF" on page 214.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rollback-information>
        <ok/>
        <configuration attributes>
           <!-- tag elements representing 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 NETCONF 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 the *CLI User Guide*.

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <rollback-information>
        <ok/>
        <configuration-information>
            <configuration-output>
                    /* formatted ASCII representing 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 is 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 NETCONF 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 NETCONF 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="URN" xmlns:junos="URL">
    <rollback-information>
        <ok/>
        <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 19.2R1 for the initial version of Junos OS Release 19.2.

Client Application

```
<rpc>
  <get-rollback-information>
    <rollback>2</rollback>
  </get-rollback-information>
</rpc>
]]>]]>
```

NETCONF Server

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
  <rollback-information>
    <ok/>
    <configuration xmlns="URL" \
        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>
]]>]]>
```

T2133

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. Prior to Junos OS Release 16.1, JSON-formatted data is requested by including the **format="json"** attribute in the opening **<get-rollback-information>** tag.

Related Documentation

- Comparing Two Previous (Rollback) Configurations Using NETCONF on page 236
- Retrieving the Rescue Configuration Using NETCONF on page 238

Comparing Two Previous (Rollback) Configurations Using NETCONF

In a NETCONF 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 NETCONF server encloses its response in **<rpc-reply>**, **<rollback-information>**, **<configuration-information>**, and **<configuration-output>** tag elements. The **<ok/>** 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 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="URN" xmlns:junos="URL">
    <rollback-information>
        <ok/>
        <configuration-information>
            <configuration-output>
               /* formatted ASCII 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>
]]>]]>
```

NETCONF Server

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

T2117

Related Documentation

- Retrieving a Previous (Rollback) Configuration Using NETCONF on page 233
- Retrieving the Rescue Configuration Using NETCONF on page 238

Retrieving the Rescue Configuration Using NETCONF

The rescue configuration is a configuration saved in case it is necessary to restore a valid, nondefault configuration. (To create a rescue configuration in a NETCONF session, use the Junos XML **<request-save-rescue-configuration>** tag element or the **request system configuration rescue save** CLI operational mode command. For more information, see the *Junos XML API Operational Developer Reference* or the CLI Explorer.)

In a NETCONF 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>** tag 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 tag elements are the default):

```
<rpc>
  <get-rescue-information/>
</rpc>
]]>]]>
```

The NETCONF server encloses its response in **<rpc-reply>**, **<rescue-information>**, and **<configuration>** tag elements. The **<ok/>** 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 Using NETCONF" on page 214.

```
  <rpc-reply xmlns="URN" xmlns:junos="URL">
      <rescue-information>
          <ok/>
          <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 NETCONF 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 the *CLI User Guide*.

```
  <rpc-reply xmlns="URN" xmlns:junos="URL">
      <rescue-information>
          <ok/>
          <configuration-information>
              <configuration-output>
                      /* formatted ASCII for 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 is 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 NETCONF 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 NETCONF 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="URN" xmlns:junos="URL">
    <rescue-information>
        <ok/>
        <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. Prior to Junos OS Release 16.1, JSON-formatted data is requested by including the **format="json"** attribute in the opening **<get-rescue-information>** tag.

Related Documentation

- Retrieving a Previous (Rollback) Configuration Using NETCONF on page 233
- Comparing Two Previous (Rollback) Configurations Using NETCONF on page 236

Requesting an XML Schema for the Configuration Hierarchy Using NETCONF

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 element, 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-config>** tag element instead, as described in "Requesting Configuration Data Using NETCONF" on page 213.

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

In a NETCONF 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 NETCONF server encloses the XML schema in **<rpc-reply>** and **<xsd:schema>** tag elements:

```
<rpc-reply xmlns="URN" xmlns:junos="URL">
    <xsd:schema>
        <!-- tag elements for the Junos XML 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 with 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 19.2R1 (and appears on two lines for legibility only):

```
<xsd:import schemaLocation="junos.xsd" \
        namespace="http://xml.juniper.net/junos/19.2R1/junos"/>
```

To enable the schema validator to interpret the **<xsd:import>** tag element, you must manually create a file called **junos.xsd** in the directory where you place the **.xsd** file that contains the complete Junos OS 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, 19.2R1 for the first release of Junos OS 19.2).

```
<?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 NETCONF 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 NETCONF server's response appear on multiple lines for legibility only. The NETCONF 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 19.2R1 for the initial version of Junos OS Release 19.2.

Client Application **NETCONF Server**

```
<rpc>
  <get-xnm-information>
    <type>xml-schema</type>
    <namespace>junos-configuration</namespace>
  </get-xnm-information>
</rpc>
]]>]]>
                        <rpc-reply  xmlns="URN" 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>
                                      .
                                      .
                                      .
```

T2114

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 **NETCONF 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> here -->
                  </xsd:choice >
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
            <!-- statements for other hierarchy levels here -->
          </xsd:choice >
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</rpc-reply>
]]>]]>
```

T2115

Related Documentation

- Requesting Configuration Data Using NETCONF on page 213
- Specifying the Source for Configuration Information Requests Using NETCONF on page 214
- Specifying the Scope of Configuration Information to Return in a NETCONF Response on page 216

PART 5

NETCONF Utilities

CHAPTER 15

NETCONF Perl Client

Understanding the NETCONF Perl Client and Sample Scripts

Devices running Junos OS support the NETCONF XML management protocol, which enables client applications to request and change configuration information on the devices. The NETCONF protocol uses an Extensible Markup Language (XML)-based data encoding for the configuration data and remote procedure calls. The Juniper Networks NETCONF Perl API enables programmers familiar with the Perl programming language to create their own Perl applications to manage devices running Junos OS over NETCONF.

NOTE: Prior to Junos OS Release 16.1, every Junos OS release included a new, release-dependent version of the NETCONF Perl client. Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The modules and sample scripts in the release-dependent versions of the NETCONF Perl distribution differ from those in the release-independent version hosted on GitHub and CPAN.

This section includes the following topics:

NETCONF Perl Client Modules

Table 6 on page 248 summarizes the modules in the release-independent version of the NETCONF Perl library. The **Net::Netconf::Manager** module provides an object-oriented interface for communicating with the NETCONF server on devices running Junos OS, and enables you to easily connect to the device, establish a NETCONF session, and execute operational and configuration requests. Client applications only directly invoke the **Net::Netconf::Manager** object. When the client application creates a **Manager** object, it supplies the device name and the login name to use when accessing the device. The login name determines the client application's access level on the device.

Table 6: NETCONF Perl Modules

Module	Description
Access	Creates an **Access** object based on the access method type specified when instantiating the object. The module is responsible for calling the **connect()** method to establish a session with the NETCONF server at the destination host and for exchanging hello packets with the server after the session is established.
Constants	Declares all NETCONF constants.
Device	Implements an object-oriented interface to the NETCONF API supported by devices running Junos OS. Objects of this class represent the local side of the connection to the device, which communicates to the client using the NETCONF protocol.
EzEditXML	Facilitates the development of XML documents for both operational and configuration requests. The module uses **XML::LibXML** as a base library, but provides Junos OS CLI-specific features to manipulate the configuration, corresponding to the CLI commands: **delete**, **activate**, **deactivate**, **insert**, and **rename**.
Manager	Instantiates and returns a NETCONF or Junos XML **Device** object depending on which server is requested.
SAXHandler	SAX-based parser that parses responses from the NETCONF server.
SSH	Provides SSH access to a **Net::Netconf::Access** instance, and manages the SSH connection with the destination host. The underlying mechanism for managing the SSH connection is based on OpenSSH.
Trace	Provides tracing levels and enables tracing based on the requested debug level.

NOTE: The following module is new in the release-independent version of the NETCONF Perl client: **EzEditXML**.

The following modules were removed in the release-independent version of the NETCONF Perl client: **Transform**, **Plugins**, and **Version**.

Client applications can also leverage Perl modules in the public domain to ease the development of NETCONF Perl client applications. Because NETCONF uses XML-based data encoding, client applications can make use of the many Perl modules that manipulate XML data.

You can use the NETCONF Perl client to create Perl applications that connect to a device, establish a NETCONF session, and execute operations. The communication between the client and the NETCONF server on the device through the NETCONF Perl API involves the following steps:

- Establishing a NETCONF session over SSHv2 between the client application and the NETCONF server on the device running Junos OS.
- Creating RPCs corresponding to requests and sending these requests to the NETCONF server.
- Receiving and processing the RPC replies from the NETCONF server.

Sample Scripts

The NETCONF Perl distribution includes an **examples** directory with the following sample scripts that illustrate how to use the modules to perform various functions. For instructions on running the scripts, see the **README** file in the NETCONF Perl GitHub repository at https://github.com/Juniper/netconf-perl.

- **diagnose_bgp/diagnose_bgp.pl**—Illustrates how to monitor the status of the device and diagnose problems. The script extracts and displays information about a device's unestablished Border Gateway Protocol (BGP) peers from the full set of BGP configuration data.
- **get_chassis_inventory/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** query with the **detail** option to request the same information as returned by the Junos XML **<get-chassis-inventory><detail/></get-chassis-inventory>** request and the CLI operational mode command **show chassis hardware detail**.
- **edit_configuration/edit_configuration.pl**—Illustrates how to configure the device by loading a file that contains configuration data formatted with Junos XML tag elements. The distribution includes a sample configuration file, **config.xml**; however, you can specify a different configuration file on the command line when you invoke the script.

Release History Table

Release	Description
16.1	Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The modules and sample scripts in the release-dependent versions of the NETCONF Perl distribution differ from those in the release-independent version hosted on GitHub and CPAN.

Related Documentation

- Installing the NETCONF Perl Client on page 249
- Writing NETCONF Perl Client Applications on page 251

Installing the NETCONF Perl Client

The Juniper Networks NETCONF Perl API enables programmers familiar with the Perl programming language to create their own Perl applications to manage and configure routing, switching, and security devices running Junos OS. The NETCONF Perl client, which is available on GitHub and through the Comprehensive Perl Archive Network (CPAN),

is independent of the Junos OS release running on the managed devices. You can use the same client installation to manage devices running any Junos OS release.

The NETCONF Perl distribution uses the same directory structure for Perl modules as CPAN. This includes a **lib** directory for the **NET::Netconf** module and its supporting files, and an **examples** directory for sample scripts. You install the NETCONF Perl distribution on a device running a Unix-like operating system. After you install the software, you can create Perl applications to connect to a device running Junos OS, establish a NETCONF session, and execute operations.

For information about installing the NETCONF Perl API, follow the instructions in the README file located in the NETCONF Perl GitHub repository at https://github.com/Juniper/netconf-perl.

NOTE: Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. Prior to Junos OS Release 16.1, every Junos OS release included a new version of the NETCONF Perl client. This release-dependent NETCONF Perl client required that you install a version of the client equal to or greater than the version of the Junos OS release running on a managed device. Doing this ensured support for all operations in that release. The release-independent distribution of the NETCONF Perl client on GitHub and CPAN removes these dependencies so that the client can manage devices running any version of the Junos OS release.

Release History Table

Release	Description
16.1	Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release.

Related Documentation

- Understanding the NETCONF Perl Client and Sample Scripts on page 247

CHAPTER 16

Developing NETCONF Perl Client Applications

Writing NETCONF Perl Client Applications

The Juniper Networks NETCONF Perl client enables programmers familiar with the Perl programming language to create their own Perl applications to manage and configure routing, switching, and security devices running Junos OS. The **Net::Netconf::Manager** module provides an object-oriented interface for communicating with a NETCONF server on devices running Junos OS, and enables you to connect to the device, establish a NETCONF session, and execute operational and configuration requests.

The following outline lists the basic tasks involved in writing a NETCONF Perl client application that manages a device running Junos OS. Each task provides a link to more detailed information about performing that task.

1. Import Perl Modules and Declare Constants—"Importing Perl Modules and Declaring Constants in NETCONF Perl Client Applications" on page 253

2. Connect to the NETCONF Server—"Connecting to the NETCONF Server in Perl Client Applications" on page 254 and "Collecting Parameters Interactively in NETCONF Perl Client Applications" on page 256

3. Submit Requests to the NETCONF Server—"Submitting a Request to the NETCONF Server in Perl Client Applications" on page 259

4. Parse and Format the Response from the NETCONF Server—"Parsing the NETCONF Server Response in Perl Client Applications" on page 269

5. Close the Connection to the NETCONF Server—"Closing the Connection to the NETCONF Server in Perl Client Applications" on page 271

The tasks are illustrated in the following example, which uses the **Net::Netconf::Manager** object to request information from a device running Junos OS. The example presents the minimum code required to execute a simple query.

NOTE: Prior to Junos OS Release 16.1, every Junos OS release included a new, release-dependent version of the NETCONF Perl client. Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The sample scripts in the release-dependent versions of the NETCONF Perl distribution differ from those in the release-independent version hosted on GitHub and CPAN.

1. Import required modules and declare constants.

```
use strict;
use Carp;
use Net::Netconf::Manager;
```

2. Create a **Manager** object and connect to the device.

```
my %deviceinfo = (
   access => "ssh",
   login => "johndoe",
   password => "password123",
   hostname => "Router1"
);
my $jnx = new Net::Netconf::Manager(%deviceinfo);

unless ( ref $jnx ) {
   croak "ERROR: $deviceinfo{hostname}: failed to connect.\n";
}
```

3. Construct the query and send it to the NETCONF server.

```
my $query = "get_chassis_inventory";
my $res = $jnx->$query();
```

4. Process the response as needed.

```
print "Server response: \n $jnx->{'server_response'} \n";
```

5. Disconnect from the NETCONF server.

```
$jnx->disconnect();
```

Release History Table

Release	Description
16.1	Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The sample scripts in the release-dependent versions of the NETCONF Perl distribution differ from those in the release-independent version hosted on GitHub and CPAN.

Related Documentation

- Understanding the NETCONF Perl Client and Sample Scripts on page 247

Importing Perl Modules and Declaring Constants in NETCONF Perl Client Applications

When creating a NETCONF Perl client application, include the following statement at the start of the application. This statement imports the functions provided by the **Net::Netconf::Manager** object, which the application uses to connect to the NETCONF server on a device.

```
use Net::Netconf::Manager;
```

Include statements to import other Perl modules as appropriate for your application. For example, several of the sample scripts included in the NETCONF Perl distribution import the following standard Perl modules, which include functions that handle input from the command line:

- Carp—Includes functions for user error warnings.
- 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 script **diagnose_bgp.pl** includes the following statement to declare a constant for the access method:

```
use constant VALID_ACCESS_METHOD => 'ssh';
```

The **edit_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

- Writing NETCONF Perl Client Applications on page 251
- Connecting to the NETCONF Server in Perl Client Applications on page 254

Connecting to the NETCONF Server in Perl Client Applications

The following sections explain how to use the **NET::Netconf::Manager** object in a Perl client application to connect to the NETCONF server on a device running Junos OS:

- Satisfy Protocol Prerequisites on page 254
- Group Requests on page 254
- Obtain and Record Parameters Required by the NET::Netconf::Manager Object on page 255
- Obtaining Application-Specific Parameters on page 255
- Establishing the Connection on page 256

Satisfy Protocol Prerequisites

The NETCONF server supports several access protocols. For each connection to the NETCONF server on a device running Junos OS, the application must specify the protocol it is using. Perl client applications can communicate with the NETCONF server via SSH only.

Before your application can run, you must satisfy the prerequisites for SSH. This involves enabling NETCONF on the device by configuring the **set system services netconf ssh** statement.

Group Requests

Establishing a connection to the NETCONF 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 device, 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 NETCONF server at a time. This is because the **NET::Netconf::Manager** object does not return control to the application until it receives the closing **</rpc-reply>** tag that represents the end of the NETCONF server's response to the current request.

Obtain and Record Parameters Required by the NET::Netconf::Manager Object

The **NET::Netconf::Manager** object takes the following required parameters, specified as keys in a Perl hash:

- **access**—The access protocol to use when communicating with the NETCONF server. Before the application runs, satisfy the SSH prerequisites.
- **hostname**—The name of the device to which to connect. For best results, specify either a fully-qualified hostname or an IP address.
- **login**—The username under which to establish the connection to the NETCONF 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**—The password corresponding to the username.

The sample scripts in the NETCONF Perl distribution record the parameters in a Perl hash called **%deviceinfo**, declared as follows:

```
my %deviceinfo = (
        'access' => $access,
        'login' => $login,
        'password' => $password,
        'hostname' => $hostname,
);
```

The sample scripts included in the NETCONF 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 NETCONF Perl Client Applications" on page 256. 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 **NET::Netconf::Manager** object, applications might need to define other parameters, such as the name of the file to which to write the data returned by the NETCONF server in response to a request.

As with the parameters required by the **NET::Netconf::Manager** 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 **NET::Netconf::Manager** object. Several examples follow.

The following line enables a debugging trace if the user includes the **-d** command-line option:

```
my $debug_level = $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 NETCONF server's response is written. If the **-o** option is not provided, the variable is set to the empty string.

```
my $outputfile = $opt{'o'} || "";
```

Establishing the Connection

After obtaining values for the parameters required for the **NET::Netconf::Manager** object, each sample script records them in the **%deviceinfo** hash.

```
my %deviceinfo = (
      'access' => $access,
      'login' => $login,
      'password' => $password,
      'hostname' => $hostname,
);
```

The script then invokes the NETCONF-specific **new** subroutine to create a **NET::Netconf::Manager** 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 Net::Netconf::Manager(%deviceinfo);
unless (ref $jnx) {
    croak "ERROR: $deviceinfo{hostname}: failed to connect.\n";
}
```

Related Documentation

- Writing NETCONF Perl Client Applications on page 251
- Importing Perl Modules and Declaring Constants in NETCONF Perl Client Applications on page 253
- Collecting Parameters Interactively in NETCONF Perl Client Applications on page 256
- Submitting a Request to the NETCONF Server in Perl Client Applications on page 259
- Closing the Connection to the NETCONF Server in Perl Client Applications on page 271

Collecting Parameters Interactively in NETCONF Perl Client Applications

In a NETCONF Perl client application, a script can interactively obtain the parameters required by the **NET::Netconf::Manager** object from the command-line.

The NETCONF Perl distribution includes several sample Perl scripts to perform various functions on devices running Junos OS. Each sample script obtains the parameters required by the **NET::Netconf::Manager** object from command-line options provided by the user who invokes the script. The scripts use the **getopts** function defined in the **Getopt::Std** Perl module to read the options from the command line and then record the options in a Perl hash called **%opt**. (Scripts used in production environments probably do not obtain parameters interactively, so this section is important mostly for understanding the sample scripts.)

The following example references the **get_chassis_inventory.pl** sample script from the NETCONF Perl GitHub repository at https://github.com/Juniper/netconf-perl/tree/master/examples/get_chassis_inventory.

NOTE: Prior to Junos OS Release 16.1, every Junos OS release included a new, release-dependent version of the NETCONF Perl client. Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The sample scripts in the release-dependent versions of the NETCONF Perl distribution differ from those in the release-independent version hosted on GitHub and CPAN.

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:d:f: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. The only supported method is 'ssh'.
  -f <xmlfile>  The name of the XML file to print server response to.
                Default: chassis_inventory.xml
  -o <filename> output is written to this file instead of standard output.
  -d <level>    Debug level [1-6]\n\n";

    croak $usage;
}
```

The **get_chassis_inventory.pl** script includes the following code to obtain values from the command line for the parameters required by the **NET::Netconf::Manager** 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, can be ssh only
my $access = $opt{'m'} || 'ssh';
use constant VALID_ACCESS_METHOD => 'ssh';
output_usage() unless (VALID_ACCESS_METHOD =~ /$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. If the specified value does not match the only valid value defined by the **VALID_ACCESSES** constant, the script invokes the **output_usage** subroutine to print the usage message.

```
my $access = $opt{'m'} || 'ssh';
use constant VALID_ACCESS_METHOD => 'ssh';
output_usage() unless (VALID_ACCESS_METHOD =~ /$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";
}
```

Release History Table

Release	Description
16.1	Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The sample scripts in the release-dependent versions of the NETCONF Perl distribution differ from those in the release-independent version hosted on GitHub and CPAN.

Related Documentation

- Writing NETCONF Perl Client Applications on page 251

Submitting a Request to the NETCONF Server in Perl Client Applications

In a NETCONF Perl client application, after establishing a connection to the NETCONF server, the client application can execute operational or configuration commands on a device running Junos OS to request operational information or change the configuration. The NETCONF Perl API supports a set of methods that correspond to CLI operational mode commands and NETCONF configuration operations. To execute a command, the client application invokes the Perl method corresponding to that command.

NOTE: Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The release-independent version of the NETCONF Perl client can invoke any method that has a corresponding Junos XML request tag.

Prior to Junos OS Release 16.1, every Junos OS release included a new, release-dependent version of the NETCONF Perl client. Each version of the software supported a set of methods that corresponded to specific CLI operational mode commands and operations on configuration objects. You can view the list of operational methods supported in that version of the client by examining the files stored in the **lib/Net/Netconf/Plugins/Plugin/*release*** directory of the NETCONF Perl distribution. The set of methods that correspond to operations on configuration objects is defined in the **lib/Net/Netconf/Plugins.pm** file of the distribution.

See the following sections for more information:

- Mapping Junos OS Commands and NETCONF Operations to Perl Methods on page 260
- Providing Method Options on page 261
- Submitting a Request on page 263

Mapping Junos OS Commands and NETCONF Operations to Perl Methods

All operational commands that have Junos XML counterparts are listed in the *Junos XML API Operational Developer Reference*. You can also display the Junos XML request tag elements for any operational mode command that has a Junos XML counterpart on the CLI. Once you obtain the request tag, you can map it to the corresponding Perl method name.

To display the Junos XML request tags for a command in the CLI, include the **| display xml rpc** option after the command. The following example displays the request tag for the **show route** command:

```
user@host> show route | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
    <rpc>
        <get-route-information>
        </get-route-information>
    </rpc>
</rpc-reply>
```

You can map the request tag for an operational command to a Perl method name. To derive the method name, replace any hyphens in the request tag with underscores, and remove the enclosing angle brackets. For example, the **<get-route-information>** request tag maps to the **get_route_information** method name.

Similarly, NETCONF protocol operations map to Perl method names in the same manner. For example, the **<edit-config>** operation maps to the **edit_config** method name.

Providing Method Options

Perl methods can have one or more options. The following section describes the notation that an application uses to define a method's options in a NETCONF Perl client application.

- A method without options is defined as **$NO_ARGS**, as in the following entry for the **get_autoinstallation_status_information** method:

```
## Method : get_autoinstallation_status_information
## Returns: <autoinstallation-status-information>
## Command: "show system autoinstallation status"
get_autoinstallation_status_information => $NO_ARGS,
```

  To invoke a method without options, the client application follows the method name with an empty set of parentheses, as in the following example:

```
$jnx->get_autoinstallation_status_information();
```

- A fixed-form option is defined as type **$TOGGLE**. In the following example, the **get_ancp_neighbor_information** method has two fixed-form options, **brief** and **detail**:

```
## Method : get_ancp_neighbor_information
## Returns: <ancp-neighbor-information>
## Command: "show ancp neighbor"
get_ancp_neighbor_information => {
    brief => $TOGGLE,
    detail => $TOGGLE,
}
```

  To include a fixed-form option when invoking a method, set the option equal to the string **'True'**, as in the following example:

```
$jnx->get_ancp_neighbor_information(brief => 'True');
```

NOTE: When using the release-dependent NETCONF Perl distribution, to include a fixed-form option when invoking a method, set the option equal to the value 1 (one).

- 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');
```

- A set of configuration statements or corresponding tag elements is defined as type **$DOM**. In the following example, the **get_config** method takes a set of configuration statements (along with two attributes):

```
'get_config' => {
     'source' => $DOM_STRING,
     'source_url' => $URL_STRING,
     'filter' => $DOM
},
```

A DOM object is XML code:

```
my $xml_string = "
<filter type=\"subtree\">
<configuration>
  <protocols>
    <bgp></bgp>
  </protocols>
</configuration>
</filter>
";

my %queryargs = (
   'source' => "running",
   'filter' => $xml_string,
);
```

This generates the following RPC request:

```
<rpc message-id='1'>
<get-config>
   <source> <running/> </source>
   <filter type="subtree">
      <configuration>
         <protocols>
            <bgp></bgp>
         </protocols>
      </configuration>
   </filter>
</get-config>
</rpc>
```

A method can have a combination of fixed-form options, options with variable values, 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 illustrates the recommended way to send a configuration request to the NETCONF server and shows how to handle error conditions. The **$jnx** variable is defined to be a **NET::Netconf::Manager** object. The sample code, which is taken from the **edit_configuration.pl** sample script, locks the candidate configuration, loads the configuration changes, commits the changes, and then unlocks the configuration database and disconnects from the NETCONF server. You can view the complete **edit_configuration.pl** script in the **examples/edit_configuration** directory in the NETCONF Perl GitHub repository at https://github.com/Juniper/netconf-perl.

```
my $res; # Netconf server response

# connect to the Netconf server
my $jnx = new Net::Netconf::Manager(%deviceinfo);
unless (ref $jnx) {
    croak "ERROR: $deviceinfo{hostname}: failed to connect.\n";
}

# Lock the configuration database before making any changes
print "Locking configuration database ...\n";
my %queryargs = ( 'target' => 'candidate' );
$res = $jnx->lock_config(%queryargs);

# See if you got an error
if ($jnx->has_error) {
    print "ERROR: in processing request \n $jnx->{'request'} \n";
    graceful_shutdown($jnx, STATE_CONNECTED, REPORT_FAILURE);
}

# 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, STATE_LOCKED, REPORT_FAILURE);
}

# Read in the XML file
my $config = read_xml_file($xmlfile);
print "\n\n$config \n\n";

%queryargs = (
        'target' => 'candidate'
     );

# If we are in text mode, use config-text arg with wrapped
```

```
# configuration-text, otherwise use config arg with raw
# XML
if ($opt{t}) {
  $queryargs{'config-text'} = '<configuration-text>' . $config
    . '</configuration-text>';
} else {
  $queryargs{'config'} = $config;
}

$res = $jnx->edit_config(%queryargs);

# See if you got an error
if ($jnx->has_error) {
    print "ERROR: in processing request \n $jnx->{'request'} \n";
    # Get the error
    my $error = $jnx->get_first_error();
    get_error_info(%$error);
    # Disconnect
    graceful_shutdown($jnx, STATE_LOCKED, REPORT_FAILURE);
}

# Commit the changes
print "Committing the <edit-config> changes ...\n";
$jnx->commit();
if ($jnx->has_error) {
    print "ERROR: Failed to commit the configuration.\n";
    graceful_shutdown($jnx, STATE_CONFIG_LOADED, REPORT_FAILURE);
}

# Unlock the configuration database and
# disconnect from the Netconf server
print "Disconnecting from the Netconf server ...\n";
graceful_shutdown($jnx, STATE_LOCKED, REPORT_SUCCESS);
```

Release History Table

Release	Description
16.1	Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The release-independent version of the NETCONF Perl client can invoke any method that has a corresponding Junos XML request tag.

Related Documentation

- Writing NETCONF Perl Client Applications on page 251
- Example: Requesting an Inventory of Hardware Components Using a NETCONF Perl Client Application on page 265
- Example: Changing the Configuration Using a NETCONF Perl Client Application on page 266
- Parsing the NETCONF Server Response in Perl Client Applications on page 269

Example: Requesting an Inventory of Hardware Components Using a NETCONF Perl Client Application

The NETCONF Perl distribution includes several sample Perl scripts to 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. It 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.

NOTE: Prior to Junos OS Release 16.1, every Junos OS release included a new, release-dependent version of the NETCONF Perl client. Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The sample scripts in the release-dependent versions of the NETCONF Perl distribution differ from those in the release-independent version hosted on GitHub and CPAN.

After establishing a connection to the NETCONF server, the script sends the **get_chassis_inventory** request and includes the **detail** argument.

```
my $query = "get_chassis_inventory";
my %queryargs = ( 'detail' => 'True' );
```

NOTE: When using the release-dependent NETCONF Perl distribution, to include a fixed-form option when invoking a method, set the option equal to the value 1 (one).

The script sends the query and assigns the return value to the **$res** variable. The script first prints the RPC request and response to standard output, then it prints the response to the specified file. The script then checks for and prints any error encountered.

```
my $res; # Netconf server response

# send the command and get the server response
my $res = $jnx->$query(%queryargs);
print "Server request: \n $jnx->{'request'}\n Server response: \n
$jnx->{'server_response'} \n";

# print the server response into xmlfile
print_response($xmlfile, $jnx->{'server_response'});

# See if you got an error
if ($jnx->has_error) {
    croak "ERROR: in processing request \n $jnx->{'request'} \n";
} else {
    print "Server Response:";
    print "$res";
```

```
}

# Disconnect from the Netconf server
$jnx->disconnect();
```

Release History Table

Release	Description
16.1	Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The sample scripts in the release-dependent versions of the NETCONF Perl distribution differ from those in the release-independent version hosted on GitHub and CPAN.

Related Documentation

- Writing NETCONF Perl Client Applications on page 251
- Submitting a Request to the NETCONF Server in Perl Client Applications on page 259

Example: Changing the Configuration Using a NETCONF Perl Client Application

The NETCONF Perl distribution includes several sample Perl scripts to perform various functions on devices running Junos OS. The **edit_configuration.pl** script locks, modifies, uploads, and commits the configuration on a device. It uses the basic structure for sending requests 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 266
- Locking the Configuration on page 267
- Reading In the Configuration Data on page 267
- Editing the Configuration Data on page 269
- Committing the Configuration on page 269

Handling Error Conditions

The **graceful_shutdown** subroutine in the **edit_configuration.pl** script handles errors encountered in the NETCONF session. It employs the following additional constants:

```
# query execution status 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 **edit_configuration.pl** script.

```
sub graceful_shutdown
{
```

```
    my ($jnx, $state, $success) = @_;
    if ($state >= STATE_CONFIG_LOADED) {
        # We have already done an <edit-config> operation
        # - Discard the changes
        print "Discarding the changes made ...\n";
        $jnx->discard_changes();
        if ($jnx->has_error) {
            print "Unable to discard <edit-config> changes\n";
        }
    }

    if ($state >= STATE_LOCKED) {
        # Unlock the configuration database
        $jnx->unlock_config();
        if ($jnx->has_error) {
            print "Unable to unlock the candidate configuration\n";
        }
    }

    if ($state >= STATE_CONNECTED) {
        # Disconnect from the Netconf server
        $jnx->disconnect();
    }

    if ($success) {
        print "REQUEST succeeded !!\n";
    } else {
        print "REQUEST failed !!\n";
    }

    exit;
}
```

Locking the Configuration

The main section of the **edit_configuration.pl** script begins by establishing a connection to a NETCONF 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 266.

```
print "Locking configuration database ...\n";
my %queryargs = ( 'target' => 'candidate' );
$res = $jnx->lock_config(%queryargs);
# See if you got an error
if ($jnx->has_error) {
    print "ERROR: in processing request \n $jnx->{'request'} \n";
    graceful_shutdown($jnx, STATE_CONNECTED, REPORT_FAILURE);
}
```

Reading In the Configuration Data

In the following code sample, the **edit_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, STATE_LOCKED, REPORT_FAILURE);
}

# Read in the XML file
my $config = read_xml_file($xmlfile);
print "\n\n$config \n\n";

%queryargs = (
                'target' => 'candidate'
             );

# If we are in text mode, use config-text arg with wrapped
# configuration-text, otherwise use config arg with raw XML
if ($opt{t}) {
  $queryargs{'config-text'} = '<configuration text> . $config .
</configuration-text>';
} else {
  $queryargs{'config'} = $config;
```

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, STATE_LOCKED, REPORT_FAILURE);
}
```

The script then invokes the **read_xml_file** subroutine, which opens the file for reading and assigns its contents to the **$config** variable. The **queryargs** key **target** is set to the value **candidate**. When the script calls the **edit_configuration** method, the candidate configuration is edited.

```
# Read in the XML file
my $config = read_xml_file($xmlfile);
print "\n\n$config \n\n";

%queryargs = (
                'target' => 'candidate'
             );
```

If the **-t** command-line option was included when the **edit_configuration.pl** script was invoked, the file referenced by the **$xmlfile** variable should contain ASCII-formatted configuration statements like those returned by the CLI configuration-mode **show** command. If the configuration statements are in ASCII-formatted text, the script encloses the configuration stored in the **$config** variable within the **configuration-text** tag element and stores the result in the value associated with the **queryargs** hash key **config-text**.

If the **-t** command-line option was not included when the **edit_configuration.pl** script was invoked, the file referenced by the **$xmlfile** variable contains Junos XML configuration

tag elements. In this case, the script stores just the **$config** variable as the value associated with the **queryargs** hash key **config**.

```
if ($opt{t}) {
  $queryargs{'config-text'} = '<configuration text> . $config .
</configuration-text>';
} else {
  $queryargs{'config'} = $config;
```

Editing the Configuration Data

The script invokes the **edit_config** method to load the configuration changes onto the device. It invokes the **graceful_shutdown** subroutine if the response from the NETCONF server has errors.

```
$res = $jnx->edit_config(%queryargs);

# See if you got an error
if ($jnx->has_error) {
    print "ERROR: in processing request \n $jnx->{'request'} \n";
    # Get the error
    my $error = $jnx->get_first_error();
    get_error_info(%$error);
    # Disconnect
    graceful_shutdown($jnx, STATE_LOCKED, REPORT_FAILURE);
```

Committing the Configuration

If there are no errors up to this point, the script invokes the **commit** method to commit the configuration on the device and make it the active configuration.

```
# Commit the changes
print "Committing the <edit-config> changes ...\n";
$jnx->commit();
if ($jnx->has_error) {
    print "ERROR: Failed to commit the configuration.\n";
    graceful_shutdown($jnx, STATE_CONFIG_LOADED, REPORT_FAILURE);
}
```

Related Documentation

- Writing NETCONF Perl Client Applications on page 251
- Submitting a Request to the NETCONF Server in Perl Client Applications on page 259

Parsing the NETCONF Server Response in Perl Client Applications

In a NETCONF Perl client application, after establishing a connection to a NETCONF server, the client application can submit one or more requests by invoking Perl methods. The NETCONF server returns the appropriate information in an **<rpc-reply>** element. There are two ways of parsing the NETCONF server's response:

- By using functions of XML::LibXML::DOM
- By using functions of XML::LibXML::XPATHContext

NOTE: Prior to Junos OS Release 16.1, every Junos OS release included a new, release-dependent version of the NETCONF Perl client. Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release. The release-independent version of the NETCONF Perl client does not include the **Net::Netconf::Transform** module that was present in the release-dependent versions of the client.

For example, consider the following reply from a NETCONF server:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/16.1R1/junos" message-id='3'>
<chassis-inventory xmlns="http://xml.juniper.net/junos/16.1R1/junos-chassis">
<chassis style="inventory">
<name>Chassis</name>
<serial-number>G1234</serial-number>
<description>MX80-48T</description>
...
</chassis>
</chassis-inventory>
</rpc-reply>
```

Suppose the user wants to parse the response and retrieve the value of the **<serial-number>** element.

The following code uses **XML::LibXMl::DOM** to retrieve the value. The example stores the response in a variable and calls methods of **DOM** to parse the response.

```
my $query = "get_chassis_inventory";
my $res = $jnx->$query();

my $rpc = $jnx->get_dom();
my $serial =
$rpc->getElementsByTagName("serial-number")->item(0)->getFirstChild->getData;

print ("\nserial number: $serial");
```

The following code uses **XML::LibXML::XPATHContext** to retrieve the value. The example stores the response in a variable and calls **XPathContext** methods to retrieve the value. The **local-name()** function returns the element name without the namespace. The XPATH expression appears on multiple lines for readability.

```
my $query = "get_chassis_inventory";
my $res = $jnx->$query();

my $rpc= $jnx->get_dom();
my $xpc = XML::LibXML::XPathContext->new($rpc);
my $serial=$xpc->findvalue('
  /*[local-name()="rpc-reply"]
  /*[local-name()="chassis-inventory"]
  /*[local-name()="chassis"]
  /*[local-name()="serial-number"]');

print ("\nserial number: $serial");
```

Release History Table

Release	Description
16.1	Beginning in Junos OS Release 16.1, the NETCONF Perl client is release-independent, is hosted on GitHub and CPAN, and can manage devices running any version of the Junos OS release.

Related Documentation

- Writing NETCONF Perl Client Applications on page 251
- Submitting a Request to the NETCONF Server in Perl Client Applications on page 259

Closing the Connection to the NETCONF Server in Perl Client Applications

In NETCONF Perl client applications, you can end the NETCONF session and close the connection to the device by invoking the **disconnect** method.

Several of the sample scripts included in the NETCONF Perl client distribution invoke the **disconnect** method in standalone statements. For example:

```
$jnx->disconnect();
```

The **edit_configuration.pl** sample script invokes the **graceful_shutdown** method, which takes the appropriate actions with regard to the configuration database and then invokes the **disconnect** method.

```
graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_SUCCESS);
```

Related Documentation

- Writing NETCONF Perl Client Applications on page 251
- Connecting to the NETCONF Server in Perl Client Applications on page 254

CHAPTER 17

NETCONF Java Toolkit

Downloading and Installing the NETCONF Java Toolkit

A *configuration management server* is a PC or workstation that is used to configure a router, switch, or security device remotely. To use the NETCONF Java toolkit, download and install the toolkit on the configuration management server. The toolkit contains the Netconf.jar library, which is compatible with Java Version 1.4 and later. The following tasks are discussed:

1. Downloading the NETCONF Java Toolkit on page 273
2. Installing the NETCONF Java Toolkit on page 273
3. Satisfying Requirements for SSHv2 Connections on page 273

Downloading the NETCONF Java Toolkit

To download the NETCONF Java toolkit to the configuration management server:

1. Access the GitHub download page at https://github.com/Juniper/netconf-java/releases .
2. Download the **Netconf.jar** file.

Installing the NETCONF Java Toolkit

To install the NETCONF Java toolkit on the configuration management server:

1. Include the **Netconf.jar** file in the CLASSPATH of your local Java development environment.
2. Ensure SSHv2/NETCONF connectivity to the device on which the NETCONF server is running.

Satisfying Requirements for SSHv2 Connections

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

The NETCONF Java toolkit accesses the NETCONF server using the SSH protocol and uses the standard SSH authentication mechanism. To establish an SSHv2 connection with a device running Junos OS, you must ensure that the following requirements are met:

- The client application has a user account and can log in to each device where a NETCONF session will be established.
- The login account used by the client application has an SSH public/private key pair or a text-based password.
- The client application can access the public/private keys or text-based password.
- The NETCONF service over SSH is enabled on each device where a NETCONF session will be established.

For information about enabling NETCONF on a device running Junos OS and satisfying the requirements for establishing an SSH session, see the *NETCONF XML Management Protocol Developer Guide*.

For information about NETCONF over SSH, see RFC 4742, *Using the NETCONF Configuration Protocol over Secure SHell (SSH)*, which is available at http://www.ietf.org/rfc/rfc4742.txt .

Related Documentation

- *Creating and Executing a NETCONF Java Application*
- *NETCONF Java Toolkit Overview*
- NETCONF XML Management Protocol and Junos XML API Overview on page 3

PART 6

YANG

CHAPTER 18

YANG Overview

Understanding YANG on Devices Running Junos OS

YANG is a standards-based, extensible data modeling language that is used to model the configuration and operational state data, remote procedure calls (RPCs), and server event notifications of network devices. The NETMOD working group in the IETF originally designed YANG to model network management data and to provide a standard for the content layer of the Network Configuration Protocol (NETCONF) model. However, YANG is protocol independent, and YANG data models can be used independent of the transport or RPC protocol and can be converted into any encoding format supported by the network configuration protocol.

Juniper Networks provides YANG modules that define the Junos OS configuration hierarchy and operational commands and Junos OS YANG extensions. You can download the YANG modules from the Juniper Networks website, from the Juniper Networks GitHub repository for YANG, or you can generate the modules on the device running Junos OS.

YANG uses a C-like syntax, a hierarchical organization of data, and provides a set of built-in types as well as the capability to define derived types. YANG stresses readability, and it provides modularity and flexibility through the use of modules and submodules and reusable types and node groups.

A YANG module defines a single data model and determines the encoding for that data. A YANG module defines a data model through its data, and the hierarchical organization of and constraints on that data. A module can be a complete, standalone entity, or it can reference definitions in other modules and submodules as well as augment other data models with additional nodes.

A YANG module defines not only the syntax but also the semantics of the data. It explicitly defines relationships between and constraints on the data. This enables you to create syntactically correct configuration data that meets constraint requirements and enables you to validate the data against the model before uploading it and committing it on a device.

YANG uses modules to define configuration and state data, notifications, and RPCs for network operations in a manner similar to how the Structure of Management Information (SMI) uses MIBs to model data for SNMP operations. However, YANG has the benefit of being able to distinguish between operational and configuration data. YANG maintains compatibility with SNMP's SMI version 2 (SMIv2), and you can use libsmi to translate SMIv2 MIB modules into YANG modules and vice versa. Additionally, when you cannot use a YANG parser, you can translate YANG modules into YANG Independent Notation (YIN), which is an equivalent XML syntax that can be read by XML parsers and XSLT scripts.

You can use existing YANG-based tools or develop custom network management applications to utilize YANG modules for faster and more accurate network programmability. For example, a client application could leverage YANG modules to generate vendor-specific configuration data for different devices and validate that data before uploading it to the device. The application could also handle and troubleshoot unexpected RPC responses and errors.

For information about YANG, see RFC 6020, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*, and related RFCs.

Related Documentation

- YANG Modules Overview on page 284
- Using Juniper Networks YANG Modules on page 293
- **show system schema on page 449**

Understanding Junos OS YANG Modules

Juniper Networks publishes the schema for devices running Junos OS using YANG models for the configuration hierarchies, operational commands, and Junos OS extensions. The following sections discuss the native Junos OS YANG modules:

- Junos OS YANG Modules Overview on page 278
- Downloading and Generating Junos OS YANG modules on page 280
- Understanding Junos OS YANG Module Namespaces and Prefixes on page 281

Junos OS YANG Modules Overview

Juniper Networks provides YANG modules that define the configuration hierarchies and operational commands, as well as YANG extensions and types, for devices running Junos OS. Starting in Junos OS Release 17.2, Junos OS YANG modules are specific to a device family. Table 7 on page 279 outlines the identifiers for the different Junos OS device families and indicates which platforms are included in each family.

Table 7: Junos OS Device Families

Device Family Identifier	Supported Platforms
junos	ACX Series, EX Series (certain platforms), MX Series, PTX Series
junos-es	J Series, LN Series, SRX Series
junos-ex	EX Series (certain platforms)
junos-qfx	QFX Series

NOTE: Different platforms within the same series might be categorized under different device families. You can verify the family for a specific device by executing the **show system information** operational mode command or the **<get-system-information/>** RPC on the device. The value of the **Family** field in the command output or the **<os-name>** element in the RPC reply indicates the device family.

Starting in Junos OS Release 17.4R1, the configuration YANG module is split into a root module that is augmented by multiple smaller modules, and the native Junos OS YANG modules use a new naming convention for the module's name, filename, and namespace. The module name and filename include the device family and the area of the configuration or command hierarchy to which the schema in the module belongs. In addition, the module filename includes a revision date. Table 8 on page 279 summarizes the YANG modules that are native to devices running Junos OS and identifies the Junos OS releases in which the different module names are used.

Table 8: Juniper Networks Native YANG Modules

Junos OS Module	Description	Module Name	Junos OS Releases
Configuration modules	Defines the schema for the Junos OS configuration hierarchy. Starting in Junos OS Release 17.4R1, the configuration YANG module is split into a root module that is augmented by multiple smaller modules.	**configuration**	14.2 through 17.3
		family*-conf-*hierarchy	17.4R1 and later
Operational command modules	Represents the operational command hierarchy and the collective group of modules that define the remote procedure calls (RPCs) for Junos OS operational mode commands. There are separate modules for the different areas of the command hierarchy.	**juniper-command**	16.1 through 17.3
		family*-rpc-*hierarchy	17.4R1 and later

Table 8: Juniper Networks Native YANG Modules (continued)

Junos OS Module	Description	Module Name	Junos OS Releases
DDL extensions module	Contains Junos OS Data Definition Language (DDL) statements for devices running Junos OS. This module includes the **must** and **must-message** keywords, which identify configuration hierarchy constraints that use special keywords. The module also includes statements that are required in custom RPCs.	**junos-extension**	15.1 through 17.3
		junos-common-ddl-extensions	17.4R1 and later
ODL extensions module	Contains Junos OS Output Definition Language (ODL) statements that can be used to create and customize formatted ASCII output for RPCs executed on devices running Junos OS.	**junos-extension-odl**	16.1 through 17.3
		junos-common-odl-extensions	17.4R1 and later
Types module	Contains definitions for Junos OS YANG types	**junos-common-types**	17.4R1 and later

To support YANG modules for different device families in different releases, the downloaded modules are organized by device family, and each module's name, filename, and namespace reflects the device family to which the schema in the module belongs. For information about obtaining the modules, see "Downloading and Generating Junos OS YANG modules" on page 280. For information about the module namespaces, see"Understanding Junos OS YANG Module Namespaces and Prefixes" on page 281.

Downloading and Generating Junos OS YANG modules

You can download the Junos OS YANG modules from the Juniper Networks website at https://www.juniper.net/support/downloads or from the Juniper Networks GitHub repository for YANG at https://github.com/Juniper/yang. You can also generate the modules on a device running Junos OS.

In Junos OS Release 17.1 and earlier, the YANG modules for the Junos OS configuration and command hierarchies that are posted on the Juniper Networks website and in GitHub define the schema for all devices running that Junos OS release. By contrast, the YANG modules generated on the local device define the schema specific to that device, including nodes both from native modules and from any standard or custom modules that have been added to the device.

Starting in Junos OS Release 17.2, Junos OS YANG modules are specific to a device family and each module's namespace reflects the device family to which the schema in the module belongs. As a result, the tar archive that is posted on the Juniper Networks website and that contains the YANG modules for a given release includes a separate directory for each device family's modules and a **common** directory for the modules that are common to all device families. Each family-specific directory uses its device family identifier as the directory name and contains the configuration and operational command modules that are supported on the platforms in that family. The device family identifiers are defined in Table 7 on page 279. The YANG modules generated on a local device running Junos OS Release 17.2 still define the schema specific to that device.

Starting in Junos OS Release 17.4R1, the YANG modules generated on a local device, by default, contain family-specific schemas, which are identical across all devices in the given device family. To generate device-specific modules, configure the **device-specific** configuration statement at the **[edit system services netconf yang-modules]** hierarchy level.

Table 9 on page 281 summarizes the scope of the schema in the downloaded and generated YANG modules for different Junos OS releases.

Table 9: Scope of Junos OS YANG Schema

Junos OS Release	Scope of Schema in Downloaded Modules	Scope of Schema in Generated Modules
17.1 and earlier	All devices	Device
17.2 through 17.3	Device family	Device
17.4R1 and later	Device family	Device family

For more information about how to download or generate the Junos OS YANG modules, see "Using Juniper Networks YANG Modules" on page 293.

Understanding Junos OS YANG Module Namespaces and Prefixes

In Junos OS Release 17.1 and earlier, Junos OS YANG modules use a unique identifier to differentiate the namespace for each module.

```
namespace "http://yang.juniper.net/yang/1.1/module-id;
```

Starting in Junos OS Release 17.2R1, the Junos OS YANG modules are specific to a device family. To support distinct YANG modules for different device families in a given Junos OS release, the YANG modules use a namespace that includes the module name, the device family, and the Junos OS release string, in addition to the identifier. For example:

```
namespace
"http://yang.juniper.net/yang/1.1/module-id/module-name/device-family/release";
```

Starting in Junos OS Release 17.4R1, the namespace is simplified to include the device family, the module type, and an identifier that is unique to each module and that differentiates the namespace of the module from that of other modules.

```
namespace "http://yang.juniper.net/device-family/type/identifier";
```

The following definitions apply to all versions of the namespace in which that variable appears:

device-family—Identifier for the device family to which the schema in the module belongs, for example, **junos**, **junos-es**, **junos-ex**, or **junos-qfx**. The different device families are outlined in Table 7 on page 279.

Modules with device-specific schemas and modules with family-specific schemas both use the same device family identifier in the namespace.

NOTE: The Junos OS common modules use the junos device family identifier in the namespace, but the modules are common to all device families.

identifier—String that differentiates the namespace of the module from that of other modules.

Junos OS configuration and command modules include an identifier that indicates the area of the configuration or command hierarchy to which the schema in the module belongs. Common modules use the module name differentiator as an identifier, for example **odl-extensions**.

module-id—Unique identifier specific to the module, for example, **jc**, **jrpc**, **je**, or **jodl**.

module-name—Name of the YANG module included in that file, for example, **configuration** or **junos-extension**. Each of the individual **juniper-command** modules uses its own unique module name in the namespace, for example, **show-class-of-service**.

release—Junos OS release in which the schema in that module is supported.

type—Type of the module. Possible values include:

- **conf**—Configuration YANG module that defines the schema for the indicated area of the configuration.
- **rpc**—Operational command YANG module that defines the RPCs for operational commands in the indicated area of the command hierarchy.
- **common**—Extension or type module that is common across all device families.

Table 10 on page 283 outlines each module's namespace URI and prefix (as defined by the module's **prefix** statement) in the different Junos OS releases. Starting in Junos OS Release 17.2, the prefix for each operational command module reflects the command hierarchy area of the RPCs included in that module. Similarly, starting in Junos OS Release 17.4R1, the prefix for each configuration YANG module reflects the configuration statement hierarchy that is included in that module. The Junos OS YANG extension and type modules use the **junos** device family identifier in the namespace, but the modules are common to all device families.

Table 10: Namespaces and Prefixes for Junos OS YANG Modules

Junos OS Module	Junos OS Release	Namespace URI	Prefix
Configuration modules	17.1 and earlier	http://yang.juniper.net/yang/1.1/jc	**jc**
	17.2 through 17.3	http://yang.juniper.net/yang/1.1/jc/configuration/*device-family/release*	**jc**
	17.4R1 and later	http://yang.juniper.net/*device-family*/conf/*hierarchy*	**jc** (root module) **jc-*hierarchy***
Operational command modules	17.1 and earlier	http://yang.juniper.net/yang/1.1/jrpc	**jrpc**
	17.2 through 17.3	http://yang.juniper.net/yang/1.1/jrpc/*module-name/device-family/release*	*hierarchy*
	17.4R1 and later	http://yang.juniper.net/*device-family*/rpc/*hierarchy*	*hierarchy*
DDL extensions module	17.1 and earlier	http://yang.juniper.net/yang/1.1/je/	**junos**
	17.2 and later	http://yang.juniper.net/yang/1.1/je/junos-extension/junos/*release*	**junos**
	17.4R1 and later	http://yang.juniper.net/junos/common/ddl-extensions	**junos**
ODL extensions module	17.1 and earlier	http://yang.juniper.net/yang/1.1/jodl	**junos-odl**
	17.2 through 17.3	http://yang.juniper.net/yang/1.1/jodl/junos-extension-odl/junos/*release*	**junos-odl**
	17.4R1 and later	http://yang.juniper.net/junos/common/odl-extensions	**junos-odl**
Types module	17.4R1 and later	http://yang.juniper.net/junos/common/types	**jt**

Starting with Junos OS Release 17.2, when you configure the **rfc-compliant** statement at the **[edit system services netconf]** hierarchy level and request configuration data in a NETCONF session, the server sets the default namespace for the **<configuration>** element to the same namespace as in the corresponding YANG model. For example:

```
<rpc>
  <get-config>
    <source>
```

```
      <running/>
    </source>
  </get-config>
</rpc>
]]>]]>

<nc:rpc-reply
  xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
  xmlns:junos="http://xml.juniper.net/junos/17.2R1/junos">
<nc:data>
<configuration
  xmlns="http://yang.juniper.net/yang/1.1/jc/configuration/junos/17.2R1.13"
  junos:commit-seconds="1493761452"
  junos:commit-localtime="2017-05-02 14:44:12 PDT"
  junos:commit-user="user">
  ...
</configuration>
</nc:data>
</nc:rpc-reply>
]]>]]>
```

Release History Table

Release	Description
17.4R1	Starting in Junos OS Release 17.4R1, the configuration YANG module is split into a root module that is augmented by multiple smaller modules, and the native Junos OS YANG modules use a new naming convention for the module's name, filename, and namespace.
17.4R1	Starting in Junos OS Release 17.4R1, the YANG modules generated on a local device, by default, contain family-specific schemas, which are identical across all devices in the given device family.
17.2R1	Starting in Junos OS Release 17.2, Junos OS YANG modules are specific to a device family and each module's namespace reflects the device family to which the schema in the module belongs.
17.2R1	Starting in Junos OS Release 17.2, the prefix for each operational command module reflects the command hierarchy area of the RPCs included in that module.

Related Documentation

- Using Juniper Networks YANG Modules on page 293
- Understanding the YANG Modules That Define the Junos OS Configuration on page 285
- Understanding the YANG Modules for Junos OS Operational Commands on page 288
- Understanding the YANG Module for Junos OS Extensions on page 291
- **show system schema on page 449**

YANG Modules Overview

YANG data models comprise modules and submodules and can define configuration and state data, notifications, and RPCs for use by NETCONF-based operations. A YANG

module defines a data model through its data, and the hierarchical organization of and constraints on that data. Each module is uniquely identified by a namespace URI.

A module defines a single data model. However, a module can reference definitions in other modules and submodules by using the **import** statement to import external modules or the **include** statement to include one or more submodules. Additionally, a module can augment another data model by using the **augment** statement to define the placement of the new nodes in the data model hierarchy and the **when** statement to define the conditions under which the new nodes are valid. A module uses the **feature** statement to specify parts of a module that are conditional and the **deviation** statement to specify where the device's implementation might deviate from the original definition.

When you import an external module, you define a prefix that is used when referencing definitions in the imported module. We recommend that you use the same prefix as that defined in the imported module to avoid conflicts.

YANG models data using a hierarchical, tree-based structure with nodes. YANG defines four nodes types. Each node has a name, and depending on the node type, the node might either define a value or contain a set of child nodes. The nodes types are:

- leaf node—Contains a single value of a specific type
- leaf-list node—Contains a sequence of leaf nodes
- container node—Contains a grouping of related nodes containing only child nodes, which can be any of the four node types
- list node—Contains a sequence of list entries, each of which is uniquely identified by one or more key leafs

In YANG, each leaf and leaf-list node includes the **type** statement to identify the data type for valid data for that node. YANG defines a set of built-in types and also provides the **typedef** statement for defining a derived type from a base type, which can be either a built-in type or another derived type.

By default, a node defines configuration data. A node defines state data if it is tagged as **config false**. Configuration data is returned using the NETCONF **<get-config>** operation, and state data is returned using the NETCONF **<get>** operation.

Related Documentation

- Understanding YANG on Devices Running Junos OS on page 277
- Using Juniper Networks YANG Modules on page 293
- **show system schema on page 449**

Understanding the YANG Modules That Define the Junos OS Configuration

Juniper Networks publishes the Junos OS configuration schema using YANG models. In Junos OS Release 17.3 and earlier, the Junos OS configuration schema is published in a single YANG module. Starting in Junos OS Release 17.4R1, the Junos OS configuration schema is published using a root configuration module that is augmented by multiple,

smaller modules. This enables consumers of the schema to only import the modules required for their tasks.

NOTE: Starting in Junos OS Release 17.4R1, Junos OS YANG modules use a new naming convention for the module's name, filename, and namespace. For more information, see "Understanding Junos OS YANG Modules" on page 278.

The root configuration module comprises the top level configuration node and any nodes that are not emitted as separate modules. Separate, smaller modules augment the root configuration module for the different configuration statement hierarchies. The configuration modules that augment the root module contain the schema for the configuration statement hierarchy level that is indicated in the module's name, filename, and namespace.

The following example shows a portion of the module containing the YANG model for the **[edit interfaces]** hierarchy:

```
/*
 * Copyright (c) 2017 Juniper Networks, Inc.
 * All rights reserved.
 */
 module junos-conf-interfaces {
   namespace "http://yang.juniper.net/junos/conf/interfaces";

   prefix jc-interfaces;

   import junos-common-types {
     prefix jt;
   }

   import junos-conf-root {
     prefix jc;
   }

   organization "Juniper Networks, Inc.";

   contact "yang-support@juniper.net";

   description "Junos interfaces configuration module";

   revision 2017-01-01 {
     description "Junos: 17.4R1.17";
   }

   augment /jc:configuration {
     uses interfaces-group;
   }

   augment /jc:configuration/jc:groups {
     uses interfaces-group;
   }
   ...
```

YANG utilities need to import only those modules required for the specific configuration task at hand. As a result, tools that consume the configuration modules require less time to compile, validate, or perform other functions on the modules than when importing a single, large module.

To determine the configuration YANG module corresponding to a specific area of the configuration, issue the **show | display detail** configuration mode command. In the following example, the schema for the **[edit protocols ospf]** hierarchy level is included in the **junos-conf-protocols@2017-01-01.yang** module.

```
user@host# show protocols ospf | display detail
##
## ospf: OSPF configuration
## YANG module: junos-conf-protocols@2017-01-01.yang
## lsa-refresh-interval: LSA refresh interval (minutes)
## range: 25 .. 50
##
## default: 50
##
...
```

You can download the Junos OS YANG modules from the Juniper Networks download site, or you can generate them on the local device. To generate the configuration modules on the local device, issue the **show system schema format yang module *module*** command. The Junos OS release determines the available command options.

- In Junos OS Release 17.3 and earlier, specify the **configuration** module.

  ```
  user@host> show system schema format yang module configuration
  ```

- In Junos OS Release 17.4 and later, specify an individual module name to return a single configuration module, or specify **all-conf** to return all configuration modules.

  ```
  user@host> show system schema format yang module all-conf output-directory /var/tmp/yang
  ```

  Starting in Junos OS Release 19.2R1, you must include the **output-directory** command option and specify the directory in which to generate the file or files. In earlier releases, you can omit the **output-directory** option when requesting a single module to display the module in standard output.

NOTE: To generate the modules from a remote session, execute the <get-yang-schema> Junos OS RPC or the <get-schema> Network Configuration Protocol (NETCONF) operation with the appropriate options.

If you specify **module configuration** or **module all-conf**, the output files include both native Junos OS configuration modules as well as any standard or custom configuration modules that have been added to the device.

NOTE: Starting in Junos OS Release 17.4R1, the native YANG modules generated on a local device contain family-specific schemas, which are identical across all devices in the given device family. In earlier releases, the generated modules contain device-specific schemas. To generate device-specific modules in Junos OS Release 17.4R1 and later, configure the **device-specific** configuration statement at the **[edit system services netconf yang-modules]** hierarchy level.

Release History Table

Release	Description
19.2R1	Starting in Junos OS Release 19.2R1, you must include the **output-directory** command option and specify the directory in which to generate the file or files.
17.4R1	Starting in Junos OS Release 17.4R1, the Junos OS configuration schema is published using a root configuration module that is augmented by multiple, smaller modules.
17.4R1	Starting in Junos OS Release 17.4R1, Junos OS YANG modules use a new naming convention for the module's name, filename, and namespace.

Related Documentation

- Using Juniper Networks YANG Modules on page 293
- Understanding Junos OS YANG Modules on page 278
- **show system schema on page 449**

Understanding the YANG Modules for Junos OS Operational Commands

Juniper Networks publishes YANG modules that define the remote procedure calls (RPCs) for Junos OS operational mode commands. Due to the large number of operational commands on devices running Junos OS, there are multiple operational command modules for each device family. There is a module for each top-level operational command group (**clear**, **file**, **monitor**, and so on) where there is at least one command within that hierarchy with an RPC equivalent. There is also a separate module for each area within the **show** command hierarchy.

NOTE: Starting in Junos OS Release 17.2, Junos OS YANG modules are specific to a device family and use a new convention for the module namespace. In addition, each of the individual operational command modules uses the command hierarchy area of the RPCs included in that module as its namespace prefix. Prior to Junos OS Release 17.2, the prefix for all operational command modules was **jrpc**.

NOTE: Starting in Junos OS Release 17.4R1, Junos OS YANG modules use a new naming convention for the module's name, filename, and namespace. For more information, see "Understanding Junos OS YANG Modules" on page 278.

The operational command modules define the RPCs corresponding to the operational commands in the command hierarchy area indicated in the filename. The following example shows a portion of the module containing the RPCs for commands in the **clear** command hierarchy:

```
user@host> file show /var/tmp/yang/junos-rpc-clear@2017-01-01.yang
/*
 * Copyright (c) 2017 Juniper Networks, Inc.
 * All rights reserved.
 */
 module junos-rpc-clear {
   namespace "http://yang.juniper.net/junos/rpc/clear";

   prefix clear;

   import junos-common-types {
     prefix jt;
   }

   organization "Juniper Networks, Inc.";

   contact "yang-support@juniper.net";

   description "Junos RPC YANG module for clear command(s)";

   revision 2017-01-01 {
     description "Junos: 17.4R1.17";
   }

   rpc clear-cli-logical-system {
     description "Clear logical system association";
     output {
       leaf output {
         type string;
       }
     }
   }
   rpc clear-cli-satellite {
     description "Clear satellite association";
     output {
       leaf output {
         type string;
       }
     }
   }
   ...
```

You can download the Junos OS YANG modules from the Juniper Networks download site, or you can generate them on the local device. To generate the operational command

YANG modules on the local device issue the **show system schema format yang module *module*** command. The Junos OS release determines the available command options.

- In Junos OS Release 17.3 and earlier, specify the **juniper-command** module to generate all of the operational command modules.

  ```
  user@host> show system schema format yang module juniper-command
  ```

  NOTE: Starting in Junos OS Release 17.1, when you generate the **juniper-command** module, the output files are placed in the current working directory, which defaults to the user's home directory. In Junos OS Release 16.2 and earlier, the output files are placed in the **/var/tmp** directory.

- In Junos OS Release 17.4R1 and later, specify an individual module name to return a single operational command module, or specify **all-rpc** to return all operational command modules.

  ```
  user@host> show system schema format yang module all-rpc output-directory /var/tmp/yang
  ```

  Starting in Junos OS Release 19.2R1, you must include the **output-directory** command option and specify the directory in which to generate the file or files. In earlier releases, you can omit the **output-directory** option when requesting a single module to display the module in standard output.

NOTE: To generate the modules from a remote session, execute the **<get-yang-schema>** Junos OS RPC or the **<get-schema>** NETCONF operation with the appropriate options.

If you specify **module juniper-command** or **module all-rpc**, the output files include both native Junos OS operational command modules as well as any standard or custom operational command modules that have been added to the device. To use an RPC in your custom YANG module, you must import the module that contains the desired RPC into your custom module.

NOTE: Starting in Junos OS Release 17.4R1, the native YANG modules generated on a local device contain family-specific schemas, which are identical across all devices in the given device family. In earlier releases, the generated modules contain device-specific schemas. To generate device-specific modules in Junos OS Release 17.4R1 and later, configure the **device-specific** configuration statement at the **[edit system services netconf yang-modules]** hierarchy level.

Release History Table

Release	Description
19.2R1	Starting in Junos OS Release 19.2R1, you must include the **output-directory** command option and specify the directory in which to generate the file or files.
17.4R1	Starting in Junos OS Release 17.4R1, Junos OS YANG modules use a new naming convention for the module's name, filename, and namespace.
17.2R1	Starting in Junos OS Release 17.2, Junos OS YANG modules are specific to a device family and use a new convention for the module namespace. In addition, each of the individual operational command modules uses the command hierarchy area of the RPCs included in that module as its namespace prefix.
17.1R1	Starting in Junos OS Release 17.1, when you generate the **juniper-command** module, the output files are placed in the current working directory, which defaults to the user's home directory

Related Documentation

- Using Juniper Networks YANG Modules on page 293
- Understanding Junos OS YANG Modules on page 278
- **show system schema on page 449**

Understanding the YANG Module for Junos OS Extensions

The Junos OS DDL extensions YANG module contains YANG extensions for devices running Junos OS. These extensions include the **must** and **must-message** statements, which identify configuration hierarchy constraints that contain special keywords. The module also contains statements that you can include in custom RPCs to define a CLI command for the RPC and to specify details about the action script to invoke when the RPC is executed or when context-sensitive help is requested for the value of a command option or configuration statement.

NOTE: **Starting in Junos OS Release 17.4, Junos OS YANG modules use a new naming convention for the module's name, filename, and namespace. The module's name and filename include the device family and Junos OS release, and the filename also includes a revision date.**

Table 11 on page 292 outlines the statements in the DDL extensions module and provides a brief description of each statement.

Table 11: Statements in the junos-extension Module

Statement Keyword	Argument Description
action-execute	Define the actions taken when you execute a custom RPC. Use the **script** substatement to define the RPC's action script, which is invoked when you execute the RPC. Starting in Junos OS Release 17.3, the **action-execute** statement is a substatement to **command**.
action-expand	Define the script that calculates and displays the possible values for a given command option or configuration statement in a custom YANG data model when a user requests context-sensitive help in the CLI. Use the **script** substatement to define the Python script that implements the logic.
command	String defining the operational command that is used to execute the corresponding RPC in the Junos OS CLI. Starting in Junos OS Release 17.3, the **command** statement includes the substatement **action-execute**, which defines the actions taken when you execute the RPC.
must	String that identifies a constraint on the configuration data. Whereas the argument for the YANG **must** statement is a string containing an XPath expression, the argument for the **junos:must** extension statement is a string containing special Junos OS syntax required for the expression of the configuration statement path. This might include special keywords such as **any**, **all**, and **unique**.
must-message	String that defines the warning message that is emitted when the constraint defined by the corresponding **junos:must** statement evaluates to false.
pattern-message	String that defines the error message emitted when the constraint defined by the corresponding **posix-pattern** statement evaluates to false.
posix-pattern	Restrict the values accepted for nodes of type string to those that match the POSIX regular expression defined in this string.
script	String specifying the name of an action script. This is a substatement of the **action-execute** or **action-expand** statement. If **script** is a substatement of **action-execute**, the script serves as the RPC's handler and is invoked when you execute the RPC. If **script** is a substatement of **action-expand**, the script is invoked when a user requests context-sensitive help in the CLI for the value of a command option or configuration statement.

Release History Table

Release	Description
17.4R1	Starting in Junos OS Release 17.4, Junos OS YANG modules use a new naming convention for the module's name, filename, and namespace.

Related Documentation

- Understanding Junos OS YANG Modules on page 278
- Using Juniper Networks YANG Modules on page 293
- Creating Custom RPCs in YANG for Devices Running Junos OS on page 317

Using Juniper Networks YANG Modules

Juniper Networks provides YANG modules that define the configuration hierarchies and operational commands, as well as YANG extensions, for devices running Junos OS. The following sections detail how to obtain Juniper Networks YANG modules and how to import them into another module:

- Obtaining Juniper Networks YANG Modules on page 293
- Importing Juniper Networks YANG Modules on page 295

Obtaining Juniper Networks YANG Modules

To obtain the Junos OS YANG modules, you can:

- download the modules from the Juniper Networks website
- download the modules from the Juniper Networks GitHub repository for YANG
- generate the modules on a device running Junos OS

In Junos OS Release 17.1 and earlier, the YANG modules for the Junos OS configuration and command hierarchies that are posted on the Juniper Networks website define the schema for all devices running that Junos OS release. Starting in Junos OS Release 17.2, Junos OS YANG modules are specific to a device family. The YANG modules download file includes a separate directory for each device family as well as a **common** directory. Each family-specific directory contains the configuration and operational command modules that are supported on the platforms in that family, and the **common** directory contains the modules that are common to all device families. For more information about the device families, see "Understanding Junos OS YANG Modules" on page 278.

When generated on the local device, the YANG modules include both native Junos OS modules as well as any standard or custom modules that have been added to the device. Starting in Junos OS Release 17.4R1, the native YANG modules generated on a local device contain family-specific schemas, which are identical across all devices in the given device family. In Junos OS Release 17.3 and earlier, the native YANG modules generated on the local device contain device-specific schemas.

To download the Juniper Networks YANG modules:

1. Access the downloads page at https://www.juniper.net/support/downloads/junos.html.
2. Select your product.
3. In the drop-down menus, select the appropriate release type and version.
4. In the Tools section, click the YANG module link.

To generate the YANG modules from the CLI on a device running Junos OS:

1. Log in to the device running Junos OS.
2. Execute the **show system schema** operational mode command and specify the module name, the YANG format, and optionally, include any desired command options.

   The module names and command options depend on the Junos OS release running on the device.

   - In Junos OS Release 15.1 and earlier, to save the output to a specific file, include the **output-file-name** option, and specify an absolute or relative path for the output file.

     ```
     user@host> show system schema module module-name format yang
       output-file-name path
     ```

   - Starting in Junos OS Release 16.1, you can save a module in a specific directory by including the **output-directory** option.

     ```
     user@host> show system schema module module-name format yang
       output-directory path
     ```

     NOTE: Starting in Junos OS Release 19.2R1, the **show system schema** command must include the **output-directory** option to specify the directory in which to generate the output files. In earlier releases, you can omit the **output-directory** option when requesting a single module to display the module in standard output.

     In Junos OS Release 16.1 through 17.3, you can specify an alternate name for the module and the filename by including the **module-name** option.

     ```
     user@host> show system schema module module-name format yang
       output-directory path module-name module-name
     ```

     NOTE: In Junos OS Release 17.3 and earlier, you can filter for specific sections of the **configuration** module by including the **filter** command option and specifying which hierarchies to return.

For a detailed list of command options, see show system schema.

To generate the modules from a remote session:

1. Connect to the device running Junos OS. For example:

```
user@server$ ssh user@host.example.net -p 830 -s netconf
```

2. Execute the **<get-yang-schema>** RPC, and specify the module or collection name, the YANG format, and the output directory.

   The module names and command options depend on the Junos OS release running on the device.

```
<rpc>
   <get-yang-schema>
      <format>yang</format>
      <identifier>all-rpc</identifier>
      <output-directory>/var/home/user</output-directory>
   </get-yang-schema>
</rpc>
```

NOTE: Starting in Junos OS Release 19.2R1, the **<get-yang-schema>** RPC must include the **<output-directory>** element to specify the directory in which to generate the output files. In earlier releases, you can omit the **output-directory** element when requesting a single module to display the module in standard output.

NOTE: You can also use the **<get-schema>** Network Configuration Protocol (NETCONF) operation to retrieve a YANG module from the device.

Importing Juniper Networks YANG Modules

You can use YANG-based tools to leverage the Juniper Networks YANG modules. If you are developing custom YANG modules, you can reference definitions in the Juniper Networks YANG modules by importing the modules into your custom module.

To import a Juniper Networks YANG module into an existing module:

1. Include the import statement, specify the module name, and assign the prefix to use with the definitions from the imported module.

```
module acme-system {
   namespace "http://acme.example.com/system";
   prefix "acme";

   import configuration {
      prefix "jc";
   }
```

```
    import junos-extension {
       prefix "junos";
    }
…
}
```

NOTE: The naming convention for the module names, filenames, namespaces, and prefixes of the native Junos OS YANG modules depends on the Junos OS release.

2. Reference definitions in the module by using the locally defined prefix, a colon, and the node identifier or keyword.

   For example, to reference the **interface** node defined in the **configuration** module, use **jc:interface**.

Release History Table

Release	Description
17.2R1	Starting in Junos OS Release 17.2, Junos OS YANG modules are specific to a device family.

Related Documentation

- Understanding YANG on Devices Running Junos OS on page 277
- Understanding Junos OS YANG Modules on page 278
- Understanding the YANG Module for Junos OS Extensions on page 291
- Understanding the YANG Modules for Junos OS Operational Commands on page 288
- **show system schema on page 449**

CHAPTER 19

Creating and Using Non-Native YANG Modules

Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS

YANG is a standards-based, extensible data modeling language that is used to model the configuration and operational state data, remote procedure calls (RPCs), and server event notifications of network devices. Devices running Junos OS enable you to load standard or custom YANG models onto the device to add data models that are not

natively supported by Junos OS but can be supported by translation. Doing this is beneficial when you want to create device-agnostic and vendor-neutral operational and configuration models that enable the same RPC or configuration to be used on different devices from one or more vendors.

When you add YANG data models that are not natively supported by devices running Junos OS, you must also supply a script that handles the translation logic between the YANG data model and Junos OS for that device. There are two types of scripts:

- *Translation scripts* are Stylesheet Language Alternative SyntaX (SLAX) or Python scripts that map the custom configuration syntax defined by the YANG model to Junos OS syntax and then load the translated data into the configuration as a transient change during the commit operation. When you load and commit configuration data in the nonnative hierarchies on those devices, Junos OS invokes the script to perform the translation and emit the transient change.
- *Action scripts* are SLAX or Python scripts that act as handlers for your custom YANG RPCs. The YANG RPC definition uses a Junos OS YANG extension to reference the appropriate action script, which is invoked when you execute the RPC.

To use custom YANG data models on devices running Junos OS, you must add the YANG modules and associated scripts to the device by issuing the **request system yang add** command. Junos OS validates the syntax of the modules and scripts, rebuilds its schema to include the new data models, and then validates the active configuration against this schema. Although the device validates the modules and scripts as you add them, we recommend that you validate the syntax prior to merging them with the Junos OS schema by first executing the **request system yang validate** command.

NOTE: In multichassis systems, you must download and add the modules and scripts to each node in the system.

NOTE: To install OpenConfig modules that are packaged as a compressed tar file, use the **request system software add** command.

When you add YANG modules and scripts to devices running Junos OS, you must associate them with a package. Packages have a unique identifier and represent a collection of related modules, translation scripts, and action scripts. You reference the package identifier if you later update modules and scripts in that package, enable or disable translation scripts associated with the package, or delete that group of modules and scripts from the device.

When you add, update, or remove YANG modules and scripts on the device by issuing the appropriate operational commands, you do not need to reboot the device in order for the changes to take effect. Newly added RPCs and configuration hierarchies are immediately available for use, and installed translation scripts are enabled by default. You can disable translation scripts in a package at any time without removing the package and associated files from the device, which can be useful when troubleshooting translation

issues. When you disable translation for a package, you can configure and commit the statements and hierarchies added by the YANG modules in that package, but the device does not translate and commit the corresponding Junos OS configuration as a transient configuration change during the commit operation.

Before installing software on a device that has one or more custom YANG data models added to it, you must remove all configuration data corresponding to the custom YANG data models from the active configuration. After the software installation is complete, add the YANG packages and corresponding configuration data back to the device, if appropriate. For more information see "Managing YANG Packages and Configurations During a Software Upgrade or Downgrade" on page 305.

Related Documentation

- Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS on page 299
- Disabling and Enabling YANG Translation Scripts on Devices Running Junos OS on page 311
- Creating Translation Scripts for YANG Configuration Models on page 308
- Creating Custom RPCs in YANG for Devices Running Junos OS on page 317

Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS

You can load custom YANG modules on devices running Junos OS to add RPCs and data models that are not natively supported by Junos OS but can be supported by translation. When you load nonnative YANG data models onto the device, you must also load any translation scripts, action scripts, and deviation modules required by those data models.

NOTE: Starting in Junos OS Release 17.3R1, when you load custom YANG data models onto the device, you do not need to explicitly load any required Junos OS extension modules. In earlier releases, you must load the Junos OS extension modules for any packages that use the modules.

Devices running Junos OS use packages to identify a collection of related YANG modules, translation scripts, and action scripts. Each package has a unique identifier. When you add YANG modules and scripts to the device, you must associate them with a new or existing package. This topic discusses how to create, update, and delete YANG packages and add or update their associated modules and scripts.

NOTE: To prevent CLI-related or configuration database errors, we recommend that you do not perform any CLI operations or change the configuration while a device is in the process of adding, updating, or deleting a YANG package and modifying the schema.

Creating a YANG Package and Adding Modules and Scripts

To validate YANG modules and scripts and add them to a new package:

1. Download the YANG modules and any necessary scripts to any directory on the device.

2. Ensure that any unsigned Python action scripts are owned by either root or a user in the Junos OS **super-user** login class and that only the file owner has write permission for the file.

   NOTE: Users can only execute unsigned Python scripts on devices running Junos OS when the script's file permissions include read permission for the first class that the user falls within, in the order of user, group, or others.

3. (Optional) Validate the syntax of the modules and scripts.

   ```
   user@host> request system yang validate action-script [scripts] module [modules]
     translation-script [scripts]
   ```

4. Create a YANG package with a unique identifier, and specify the file paths for the modules and scripts that are part of that package, as well as for any deviation modules that identify deviations for the modules in that package.

   ```
   user@host> request system yang add package package-name module [modules]
     deviation-module [modules] translation-script [scripts] action-script [scripts]
   ```

   NOTE: You can specify the absolute or relative path to a single file, or you can add multiple files by specifying a space-delimited list of file paths enclosed in brackets.

NOTE: To install OpenConfig modules that are packaged as a compressed tar file, use the **request system software add** command. OpenConfig modules and scripts that are installed by issuing the **request system software add** command are always associated with the package identifier **openconfig**.

NOTE: Starting in Junos OS Release 18.3R1, adding, deleting, or updating YANG packages in configuration mode with the **run** command is not supported.

5. When the system prompts you to restart the Junos OS CLI, press **Enter** to accept the default value of **yes**.

```
...
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes)

Restarting cli ...
```

6. Verify that the package was created and contains the correct modules and scripts.

```
user@host> show system yang package package-name
Package ID            :package-name
YANG Module(s)        :modules
Action Script(s)      :action scripts
Translation Script(s) :translation scripts
Translation script status is enabled
```

7. If the package includes translation scripts or action scripts that are written in Python, configure the **language python** statement.

```
[edit]
user@host# set system scripts language python
user@host# commit
```

8. On multichassis systems, repeat steps 1 through 7 on each node in the system.

When you create a new package, the device stores copies of the module and script files in a new location. The device also stores copies of the action script and translation script files under the **/var/db/scripts/action** and **/var/db/scripts/translation** directories, respectively. After the modules and scripts are validated and added to the device, Junos OS rebuilds its schema to include the new data models and then validates the active

configuration against this schema. Newly added RPCs and configuration hierarchies are immediately available for use.

NOTE: Devices that use the ephemeral configuration database will delete all ephemeral configuration data in the process of rebuilding the schema.

NOTE: Junos OS does not support using **configure private** mode to configure statements corresponding to third-party YANG data models, for example, OpenConfig or custom YANG data models.

Updating a YANG Package with New or Modified Modules and Scripts

You create a new YANG package by executing the **request system yang add** command. To update an existing package to either add new modules and scripts to the package or update existing modules and scripts in the package, you must use the **request system yang update** command.

To update a YANG package with new or modified modules and scripts:

1. Download the modules and scripts to any directory on the device.

2. Ensure that any unsigned Python action scripts are owned by either root or a user in the Junos OS **super-user** login class and that only the file owner has write permission for the file.

   NOTE: Users can only execute unsigned Python scripts on devices running Junos OS when the script's file permissions include read permission for the first class that the user falls within, in the order of user, group, or others.

3. (Optional) Validate the syntax of the modules and scripts.

   ```
   user@host> request system yang validate action-script [scripts] module [modules]
     translation-script [scripts]
   ```

4. Update the YANG package by issuing the **request system yang update** command, and specify the file paths for the new and modified modules and scripts.

   ```
   user@host> request system yang update package-name module [modules]
     deviation-module [modules] translation-script [scripts] action-script [scripts]
   ```

   NOTE: You can specify the absolute or relative path to a single file, or you can update multiple files by specifying a space-delimited list of file paths enclosed in brackets.

NOTE: Starting in Junos OS Release 18.3R1, adding, deleting, or updating YANG packages in configuration mode with the run command is not supported.

5. When the system prompts you to restart the Junos OS CLI, press **Enter** to accept the default value of **yes**.

```
...
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes)

Restarting cli ...
```

6. If the package includes translation scripts or action scripts that are written in Python, configure the **language python** statement, if it is not already configured.

```
[edit]
user@host# set system scripts language python
user@host# commit
```

7. On multichassis systems, repeat steps 1 through 6 on each node in the system.

When you update a package, the device stores copies of the new and modified module and script files. Junos OS then rebuilds its schema to include any changes to the data models associated with that package and validates the active configuration against this schema.

NOTE: Devices that use the ephemeral configuration database will delete all ephemeral configuration data in the process of rebuilding the schema.

Deleting a YANG Package

CAUTION: Before you delete a YANG package from a device running Junos OS, ensure that the active configuration does not contain configuration data that has dependencies on the data models added by that package.

To delete a YANG package and all modules and scripts associated with that package from a device running Junos OS:

1. Review the active configuration to determine if there are any dependencies on the YANG modules that will be deleted.

2. If the configuration contains dependencies on the modules, update the configuration to remove the dependencies.

3. Delete the package and associated modules and scripts by issuing the **request system yang delete** command with the appropriate package identifier.

   ```
   user@host> request system yang delete package-name
   ```

   ```
   Building schema and reloading /config/juniper.conf.gz ...
   Activating /config/juniper.conf.gz ...
   mgd: commit complete
   Restarting mgd ...
   ```

   NOTE: You must use the request system software delete command to remove OpenConfig packages that were installed from a compressed tar file by issuing the request system software add command.

   NOTE: Starting in Junos OS Release 18.3R1, adding, deleting, or updating YANG packages in configuration mode with the run command is not supported.

4. If the system prompts you to restart the Junos OS CLI, press **Enter** to accept the default value of **yes**.

   ```
   Building schema and reloading /config/juniper.conf.gz ...
   Activating /config/juniper.conf.gz ...
   mgd: commit complete
   Restarting mgd ...

   WARNING: cli has been replaced by an updated version:
   ...
   Restart cli using the new version ? [yes,no] (yes)

   Restarting cli ...
   ```

When you delete a package, Junos OS rebuilds its schema to remove the data models associated with that package and then validates the active configuration against this schema. The device removes the copies of the module and script files that were generated when the package was created. The device also removes the copies of the package's

action script and translation script files that are stored under the **/var/db/scripts/action** and **/var/db/scripts/translation** directories. If you downloaded the original module and script files to a different location, the original files remain unchanged.

NOTE: Devices that use the ephemeral configuration database will delete all ephemeral configuration data in the process of rebuilding the schema.

Release History Table

Release	Description
18.3R1	Starting in Junos OS Release 18.3R1, adding, deleting, or updating YANG packages in configuration mode with the **run** command is not supported.
17.3R1	Starting in Junos OS Release 17.3R1, when you load custom YANG data models onto the device, you do not need to explicitly load any required Junos OS extension modules.

Related Documentation

- Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS on page 297
- Managing YANG Packages and Configurations During a Software Upgrade or Downgrade on page 305
- **request system yang add on page 436**
- **request system yang delete on page 439**
- **request system yang update on page 445**
- **show system yang package on page 453**

Managing YANG Packages and Configurations During a Software Upgrade or Downgrade

Certain devices running Junos OS enable you to load custom YANG modules on the device to add data models that are not natively supported by Junos OS. When you add, update, or delete a YANG data model, Junos OS rebuilds its schema and then validates the active configuration against the updated schema.

When you upgrade or downgrade Junos OS, by default, the system validates the software package or bundle against the current configuration. During the installation, the schema for custom YANG data models is not available. As a result, if the active configuration contains dependencies on these models, the software validation fails, which causes the upgrade or downgrade to fail.

In addition, devices that are running Junos OS based on FreeBSD version 6 remove custom YANG packages from the device during the software installation process. For this Junos OS variant, if the active configuration contains dependencies on custom YANG data models, the software installation fails even if you do not validate the software against the configuration, because the configuration data cannot be validated during the initial boot-time commit.

For these reasons, before you upgrade or downgrade the Junos OS image on a device that has one or more custom YANG modules added to it, you must remove all configuration data corresponding to the custom YANG data models from the active configuration. After the software installation is complete, add the YANG packages and corresponding configuration data back to the device, if appropriate. The tasks are outlined in this topic.

NOTE: You do not need to delete configuration data corresponding to OpenConfig packages before upgrading or downgrading Junos OS.

- Backing up and Deleting the Configuration Data on page 306
- Restoring the YANG Packages and Configuration Data on page 307

Backing up and Deleting the Configuration Data

If the configuration contains dependencies on custom YANG data models:

1. If you plan to restore the configuration data that corresponds to the nonnative YANG data models after the software is updated, save a copy of either the entire configuration or the configuration data corresponding to the YANG data models, as appropriate.
   - To save the entire configuration:

     ```
     user@host> show configuration | save (filename | url)
     ```

   - To save configuration data under a specific hierarchy level:

     ```
     user@host> show configuration path-to-yang-statement-hierarchy | save (filename
       | url)
     ```

2. In configuration mode, delete the portions of the configuration that depend on the custom YANG data models.

   ```
   [edit]
   user@host# delete path-to-yang-statement-hierarchy
   ```

3. Commit the changes.

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

4. Prior to performing the software installation, ensure that the saved configuration data and the YANG module and script files are saved to a local or remote location that will preserve the files during the installation and that will be accessible after the installation is complete.

Restoring the YANG Packages and Configuration Data

After the software installation is complete, load the YANG packages onto the device (where required), and restore the configuration data associated with the packages, if appropriate. During a software upgrade or downgrade, devices running Junos OS with upgraded FreeBSD preserve custom YANG packages, whereas devices running Junos OS based on FreeBSD version 6 delete the packages.

1. Load the YANG packages (devices running Junos OS based on FreeBSD version 6 only).

   ```
   user@host> request system yang add package package-name module [modules]
     deviation-module [modules] translation-script [scripts] action-script [scripts]
   ```

2. When the system prompts you to restart the Junos OS CLI, press **Enter** to accept the default value of **yes**.

   ```
   ...
   WARNING: cli has been replaced by an updated version:
   ...
   Restart cli using the new version ? [yes,no] (yes)

   Restarting cli ...
   ```

   NOTE: To prevent CLI-related or configuration database errors, we recommend that you do not perform any CLI operations or change the configuration while a device is in the process of adding, updating, or deleting a YANG package and modifying the schema.

3. In configuration mode, load the configuration data associated with the YANG packages.

   For example, to load the configuration data from a file relative to the top level of the configuration statement hierarchy:

   ```
   [edit]
   user@host# load merge (filename | url)
   ```

   NOTE: For more information about loading configuration data, see the *CLI User Guide*.

4. Commit the changes.

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

Related Documentation

- Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS on page 299

Creating Translation Scripts for YANG Configuration Models

You can load YANG modules on devices running Junos OS to add data models that are not natively supported by Junos OS but that can be supported by translation. When you extend the configuration hierarchy with nonnative YANG data models, you must also supply one or more translation scripts that provide the logic to map the nonnative configuration syntax to the corresponding Junos OS syntax.

Translation scripts convert the configuration data corresponding to the nonnative YANG data models into Junos OS syntax and add the translated configuration data as a transient change in the checkout configuration during the commit operation. Translation scripts can be written in either Python or SLAX and are similar to commit scripts in structure. For information about creating SLAX and Python scripts that generate transient changes in the configuration, see the *Automation Scripting Feature Guide*.

You use the **request system yang add** or **request system yang update** commands to add YANG modules and their associated translation scripts to a new or existing YANG package on the device. After you add the modules and translation scripts to the device, you can configure the statements and hierarchies in the data model added by those modules. When you load and commit the configuration data, the device calls the script to perform the translation and generate the transient configuration change.

NOTE: Junos OS does not support using **configure private** mode to configure statements corresponding to third-party YANG data models, for example, OpenConfig or custom YANG data models.

This topic discusses the general structure for translation scripts. The specific translation logic required in the actual script depends on the custom hierarchies added to the schema and is beyond the scope of this topic.

To create the framework for translation scripts that are used on devices running Junos OS:

1. In your favorite editor, create a new file that uses the **.slax** or **.py** file extension, as appropriate.

2. Include the necessary boilerplate required for that script's language, which is identical to the boilerplate for commit scripts, and also include any required namespace declarations for your data models.

   - SLAX code:

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

```
ns prefix = "namespace";
import "../import/junos.xsl";

match configuration {
   /*
    * insert your code here
    */
}
```

- Python code:

```
from junos import Junos_Context
from junos import Junos_Configuration
import jcs

if __name__ == '__main__':
   /*
    * insert your code here
    */
```

NOTE: Translation scripts must fully qualify identifiers for nonnative YANG data models in the translation code.

NOTE: For information about commit script boilerplate code, see *Required Boilerplate for Commit Scripts* and the *Automation Scripting Feature Guide*.

3. Add code that maps the nonnative configuration data into the equivalent Junos OS syntax and stores the translated configuration data in a variable.

   - SLAX sample code:

```
match configuration {

    /* translation code */

    var $final = {
       /*
        * translated configuration
        */
    }
}
```

   - Python sample code:

```
if __name__ == '__main__':

    /* translation code */

    final = """
        <transient-change>
           /*
             * Junos XML elements representing translated configuration
```

```
            */
        </transient-change>
    """
```

4. Add the translated content to the checkout configuration as a transient configuration change by calling the **jcs:emit-change()** template in SLAX scripts or the **jcs.emit_change()** function in Python scripts with the translated configuration and **transient-change** tag as arguments.

   - SLAX sample code:

```
match configuration {

    /* translation code */

    var $final = {
       /*
        * translated configuration
        */
    }
    call jcs:emit-change($content=$final, $tag='transient-change');
}
```

   - Python sample code:

```
if __name__ == '__main__':

    /* translation code */

    final = """
           /*
            * Junos XML elements representing translated configuration
            */
    """
    jcs.emit_change(final, "transient-change", "xml")
```

NOTE: In SLAX scripts, you can also generate the transient change by emitting the translated configuration inside of a <transient-change> element instead of calling the jcs:emit-change() template.

On the device, perform the following tasks before adding the translation script to a YANG package:

1. If the translation script is written in Python, configure the **language python** statement.

```
[edit]
user@host# set system scripts language python
```

2. Download the script to the device, and optionally validate the syntax.

```
user@host> request system yang validate translation-script script
```

Before you can use translation scripts on a device, you must add the scripts and associated modules to a new or existing YANG package by issuing the **request system yang add** or **request system yang update** command. After the modules and scripts are added, the translation scripts are automatically invoked when you commit configuration data in the corresponding data models.

The active and candidate configurations contain the configuration data for the nonnative YANG data models in the syntax defined by those models. However, because the translated configuration data is committed as a transient change, the active and candidate configurations do not explicitly display the translated data in the Junos OS syntax when you issue the **show** or **show configuration** commands. To apply YANG translation scripts when you view the configuration, use the **| display translation-scripts** filter.

To view the complete post-inheritance configuration with the translated data (transient changes) explicitly included , append the **| display translation-scripts** filter to the **show configuration** command in operational mode or the **show** command in configuration mode. To view just the nonnative configuration data after translation, use the **| display translation-scripts translated-config** filter.

In configuration mode, to display just the changes to the configuration data corresponding to nonnative YANG data models before or after translation scripts are applied, append the **configured-delta** or **translated-delta** keyword, respectively, to the **show | display translation-scripts** command. In both cases, the XML output displays the deleted configuration data, followed by the new configuration data.

For more information about the **| display translation-scripts** filter, see "Committing and Displaying Configuration Data for Nonnative YANG Modules" on page 313.

Related Documentation

- Disabling and Enabling YANG Translation Scripts on Devices Running Junos OS on page 311
- Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS on page 297

Disabling and Enabling YANG Translation Scripts on Devices Running Junos OS

You can load standard (IETF, OpenConfig) or custom YANG data models on devices running Junos OS to add data models that are not natively supported by Junos OS but can be supported by translation. When you extend the configuration hierarchy with nonnative data models, you must also supply one or more translation scripts; these map the custom configuration syntax defined by the YANG data model to the corresponding Junos OS syntax and add the translated data to the checkout configuration as a transient change during the commit operation. Translation scripts are enabled by default as soon as you add them to the device by issuing the appropriate operational command.

You can disable the translation scripts in a YANG package at any time without removing the package and associated files from the device, which can be useful when troubleshooting translation issues. After you disable translation for a package and commit the configuration, the configuration data associated with the YANG data models in that

package can be present in the active configuration, but the configuration has no impact on the functioning of the device.

When translation is disabled, you can still configure and commit the statements and hierarchies in the data models added by that package. However, the device does not commit the corresponding Junos OS configuration statements as transient changes during the commit operation for any statements in the data models added by that package, even for those statements that were committed prior to disabling translation.

To disable translation scripts for a given YANG package that is installed on a device running Junos OS:

1. Issue the **request system yang disable** command, and specify the package identifier.

   ```
   user@host> request system yang disable package-name
   ```

2. Verify that the status of the translation scripts in the package is **disabled**.

   ```
   user@host> show system yang package package-name
   Package ID            :package-name
   YANG Module(s)        :modules
   Translation Script(s) :translation scripts
   Translation script status is disabled
   ```

NOTE: When you disable translation for a package, the device retains any transient configuration changes that were committed prior to disabling translation until the next commit operation.

NOTE: In configuration mode, you can issue the **show | display translation-scripts translated-config** command to verify which configured statements from nonnative YANG data models will be translated and committed during a **commit** operation. The command output does not include (and the device does not commit) the corresponding Junos OS configuration for those data models for which translation has been disabled.

To enable translation scripts for a given YANG package that is installed on a device running Junos OS:

1. Issue the **request system yang enable** command, and provide the package identifier.

   ```
   user@host> request system yang enable package-name
   ```

2. Verify that the status of the translation scripts in the package is **enabled**.

   ```
   user@host> show system yang package package-name
   ```

```
Package ID            :package-name
YANG Module(s)        :modules
Translation Script(s) :translation scripts
Translation script status is enabled
```

Related Documentation

- Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS on page 297
- request system yang disable on page 441
- request system yang enable on page 443
- show system yang package on page 453

Committing and Displaying Configuration Data for Nonnative YANG Modules

You can load standardized or custom YANG modules onto devices running Junos OS to add data models that are not natively supported by Junos OS but can be supported by translation. When you extend the configuration hierarchy with new data models, you must also supply one or more translation scripts that provide the translation logic to map the nonnative configuration syntax to Junos OS. Translation scripts are enabled by default as soon as you issue the **request system yang add** or **request system yang update** command to add them to the device.

You configure nonnative data models in the candidate configuration using the syntax defined for those models. When you commit the configuration, the translation scripts translate those portions of the data and commit the corresponding Junos OS configuration as a transient change in the checkout configuration.

NOTE: Junos OS does not support using **configure private** mode to configure statements corresponding to third-party YANG data models, for example, OpenConfig or custom YANG data models.

NOTE: Starting in Junos OS Release 16.1R2, XPath expression evaluations for the following YANG keywords are disabled by default during commit operations: **leafref**, **must**, and **when**. Prior to Junos OS Release 16.1R2, Junos OS evaluates the constraints for these keywords, which can result in longer commit times.

The candidate and active configurations contain the configuration data for nonnative YANG data models in the syntax defined by those models. However, because the translated configuration data is committed as a transient change, the candidate and active configurations do not explicitly display the translated data in the Junos OS syntax when you view the configuration by using commands such as **show** or **show configuration**.

You can explicitly display the translated data in Junos OS syntax in the candidate or active configuration by appending the **| display translation-scripts** filter to the **show** command in configuration mode or the **show configuration** command in operational mode. Applying the filter displays the post-inheritance configuration with the translated configuration data from all enabled translation scripts included.

NOTE: You can only apply the **| display translation-scripts** filter to the complete Junos OS configuration. You cannot filter subsections of the configuration hierarchy.

In operational mode, issue the following command to view the committed configuration with translation scripts applied:

```
user@host> show configuration | display translation-scripts
```

Similarly, in configuration mode, issue the following command to view the candidate configuration with translation scripts applied:

```
[edit]
user@host# show | display translation-scripts
```

The output, which is truncated in this example, displays the complete post-inheritance configuration and includes the nonnative configuration data as well as the translation of that data.

```
## Last changed: 2016-05-13 16:37:42 PDT
version "16.1R1;
system {
    host-name host;
    domain-name example.com;
    ...
    /* Translated data */
    scripts {
        op {
          file test.slax;
        }
    }
    ...
}
...
/* Nonnative configuration data */
myconfig:myscript {
    op {
        filename test.slax;
    }
}
```

Alternatively, you can view just the translated portions of the hierarchy corresponding to nonnative YANG data models by appending the **translated-config** keyword to the **| display translation-scripts** filter. In operational mode, the **translated-config** keyword returns the translated data for nonnative YANG data models present in the committed configuration.

In configuration mode, the **translated-config** keyword returns the translated data for nonnative YANG data models present in the candidate configuration, which includes both committed and uncommitted configuration data.

```
user@host> show | display translation-scripts translated-config
```

```
system {
    scripts {
        op {
            file test.slax;
        }
    }
}
```

The candidate configuration reflects the configuration data that has been configured, but not necessarily committed, on the device. In configuration mode, to display just the configuration differences in the hierarchies corresponding to nonnative YANG data models before or after translation scripts are applied, append the **configured-delta** or **translated-delta** keyword, respectively, to the **show | display translation-scripts** command. In both cases, the XML output displays the deleted configuration data, followed by the new configuration data.

For example, to view the uncommitted configuration changes for the nonnative data models in the syntax defined by those data models, issue the **show | display translation-scripts configured-delta** command in configuration mode.

```
[edit]
user@host# show | display translation-scripts configured-delta
```

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/16.1R1/junos">
    <configuration operation="delete">
    </configuration>
    <configuration operation="create">
        <myscript xmlns="http://jnpr.net/yang/myscript" operation="create">
            <op>
                <filename>test2.slax</filename>
            </op>
        </myscript>
    </configuration>
    <cli>
        <banner>[edit]</banner>
    </cli>
</rpc-reply>
```

To view the uncommitted configuration changes for the nonnative data models after translation into Junos OS syntax, issue the **show | display translation-scripts translated-delta** command in configuration mode. For example:

```
[edit]
```

```
user@host# show | display translation-scripts translated-delta
```

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/R1/junos">
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos">
        <system>
              <scripts>
                <op>
                  <file>
                    <name>test2.slax</name>
                  </file>
                </op>
              </scripts>
        </system>
    </configuration>
    <!-- EOF -->
    <cli>
        <banner>[edit]</banner>
    </cli>
</rpc-reply>
```

In configuration mode, you can better understand the transient changes that will be committed for the nonnative data models by using the various filters. To verify all Junos OS statements that will be committed as transient changes by translation scripts during the **commit** operation, issue the **show | display translation-scripts translated-config** command before committing the candidate configuration. To verify the Junos OS statements that will be committed for just the changed configuration data, issue the **show | display translation-scripts translated-delta** command. If you disable translation scripts for a package, the output for these commands does not include (and the device does not commit) the corresponding Junos OS configuration for those data models for which translation has been disabled.

NOTE: Even though nonnative configuration data might be committed in the active configuration, it does not guarantee that the corresponding translated configuration is also committed as a transient change. If you disable translation and then commit nonnative configuration data, the nonnative data is present in the committed configuration. However, the device does not commit the corresponding Junos OS configuration statements as transient changes during the commit operation for any statements in the data models added by that package, even for those statements that were committed prior to disabling translation.

Table 12 on page 317 summarizes the different filters you can apply to the committed and candidate configurations when they contain configuration data corresponding to nonnative YANG data models. The table indicates the CLI mode for each filter, and the scope and syntax of the output. By selecting different filters, you can view the entire configuration, the translated portions of the configuration, or the uncommitted configuration changes, and you can view the configuration data both before and after processing by translation scripts. In configuration mode, this enables you to better determine the Junos OS changes that will be committed for the nonnative hierarchies.

Table 12: | display translation-scripts Command

Filter	Mode	Description	Syntax and Format of Output
| display translation-scripts	Operational	Return the complete, post-inheritance committed configuration and include the translation of the nonnative data into Junos OS syntax.	YANG data model and Junos OS syntax as ASCII text
	Configuration	Return the complete, post-inheritance candidate configuration and include the translation of the nonnative data into Junos OS syntax.	YANG data model and Junos OS syntax as ASCII text
| display translation-scripts translated-config	Operational	Return the translated data corresponding to all nonnative YANG data models in the committed configuration.	Junos OS ASCII text
	Configuration	Return the translated data corresponding to all nonnative YANG data models in the candidate configuration.	Junos OS ASCII text
| display translation-scripts configured-delta	Configuration	Return the uncommitted changes in the candidate configuration corresponding to nonnative YANG data models in the syntax defined by that model.	YANG data model XML
| display translation-scripts translated-delta	Configuration	Return the uncommitted changes in the candidate configuration corresponding to nonnative YANG data models after translation into Junos OS syntax.	Junos OS XML

Release History Table

Release	Description
16.1R2	Starting in Junos OS Release 16.1R2, XPath expression evaluations for the following YANG keywords are disabled by default during commit operations: **leafref**, **must**, and **when**. Prior to Junos OS Release 16.1R2, Junos OS evaluates the constraints for these keywords, which can result in longer commit times.

Creating Custom RPCs in YANG for Devices Running Junos OS

Juniper Networks provides YANG modules that define the remote procedure calls (RPCs) for Junos OS operational commands. Starting in Junos OS Release 16.1R3, you can also

create YANG data models that define custom RPCs for supported devices running Junos OS. Creating custom RPCs enables you to precisely define the input parameters and operations and the output fields and formatting for your specific operational tasks on those devices. When you extend the operational command hierarchy with a custom YANG RPC, you must also supply an action script that serves as the handler for the RPC. The RPC definition references the action script, which is invoked when you execute the RPC.

This topic outlines the general steps for creating a YANG module that defines a custom RPC for devices running Junos OS. For information about creating an RPC action script and customizing the RPC's CLI output see "Creating Action Scripts for YANG RPCs on Devices Running Junos OS" on page 324 and "Understanding Junos OS YANG Extensions for Formatting RPC Output" on page 345.

This section presents a generic template for a YANG module that defines an RPC for devices running Junos OS. The template is followed by a detailed explanation of the different sections and statements in the template.

```
module module-name {
   namespace "namespace";
   prefix prefix;

   import junos-extension {
      prefix junos;
   }
   import junos-extension-odl {
      prefix junos-odl;
   }

   organization
      "organization";
   description
      "module-description";

   rpc rpc-name {
      description "RPC-description";

      junos:command "cli-command" {
         junos:action-execute {
            junos:script "action-script-filename";
         }
      }

      input {
         leaf input-param1 {
            type type;
            description description;
         }
         leaf input-param2 {
            type type;
            description description;
         }
         // additional leaf definitions
      }
      output {
         container output-container-name {
```

```
            container container-name {
               leaf output-param1 {
                  type type;
                  description description;
                  // optional formatting statements
               }
               // additional leaf definitions

               junos-odl:format container-name-format {
                  // CLI formatting for the parent container
               }
            }

            // Additional containers
         }
      }
   }
}
```

RPCs are defined within modules. The module name should be descriptive and indicate the general purpose of the RPCs that are included in that module, and the module namespace must be unique.

```
module module-name {
   namespace "namespace";
   prefix prefix;

}
```

NOTE: As per RFC 6020, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*, the module name and the base name of the file in which the module resides must be identical. For example, if the module name is get-if-info, the module's filename must be get-if-info.yang.

The module must import the Junos OS DDL extensions module and define a prefix. The extensions module includes YANG extensions that are required in the definition of RPCs executed on devices running Junos OS.

```
   import junos-extension {
      prefix junos;
   }
```

NOTE: Starting in Junos OS Release 17.4R1, the Junos OS YANG modules use a new naming convention for the module's name, filename, and namespace.

If any of the RPCs in the module render formatted ASCII output, the module must import the Junos OS ODL extensions module and define a prefix. The ODL extensions module defines YANG extensions that you use to precisely specify how to render the output when you execute the operational command for that RPC in the CLI or when you request the RPC output in text format.

```
import junos-extension-odl {
   prefix junos-odl;
}
```

Include the organization responsible for the module as well as a description of the module.

```
organization
   "organization";
description
   "module-description";
```

Within the module, you can define one or more RPCs, each with a unique name. The RPC name is used to remotely execute the RPC, and thus should clearly indicate the RPC's purpose. The RPC purpose can be further clarified in the **description** statement. If you also define a CLI command for the RPC, the CLI displays the RPC description in the context-sensitive help for that command listing.

```
rpc rpc-name {
   description "RPC-description";

}
```

Within the RPC definition, define the **command**, **action-execute**, and **script** statements, which are Junos OS DDL extension statements. The **command** statement defines the operational command that you use to execute the RPC in the Junos OS CLI. To execute the RPC remotely, use the RPC name for the request tag.

The **action-execute** statement and **script** substatement must be defined for every RPC. The **script** substatement defines the name of the action script that is invoked when you execute the RPC. You must define one and only one action script for each RPC.

NOTE: Starting in Junos OS Release 17.3, the **action-execute** statement is a substatement to **command**. In earlier releases, the **action-execute** and **command** statements are placed at the same level, and the **command** statement is optional.

- In Junos OS Release 17.3 and later, define the **command** statement and its substatements.

```
junos:command "cli-command" {
   junos:action-execute {
      junos:script "action-script-filename";
   }
}
```

- In Junos OS Release 17.2 and earlier, define the **action-execute** and **script** statements, and optionally define the **command** statement.

```
junos:command "cli-command";
junos:action-execute {
   junos:script "action-script-filename";
}
```

NOTE: You must add the YANG module and action script to the device as part of a new or existing YANG package by issuing the **request system yang add** or **request system yang update** command. Thus, you only need to provide the name and not the path of the action script for the **junos:script** statement.

NOTE: If your action script is written in Python, you must configure the **[edit system scripts language python]** statement on each device where the script will be executed.

Input parameters to the RPC operation are defined within the optional **input** statement. When you execute the RPC, Junos OS invokes the RPC's action script and passes all of the input parameters to the script.

```
input {
   leaf input-param1 {
      type type;
      description description;
   }
   leaf input-param2 {
      type type;
      description description;
   }
   // additional leaf definitions
}
```

NOTE: Starting in Junos OS Release 19.2R1, custom YANG RPCs support input parameters of type **empty** when executing the RPC's command in the Junos OS CLI. In earlier releases, input parameters of type **empty** are only supported when executing the RPC in a NETCONF or Junos XML protocol session.

The optional **output** statement encloses the output parameters to the RPC operation. The **output** statement can include one top-level root container. It is a good practice to correlate the name of the root container and the RPC name. For example, if the RPC name is **get-xyz-information**, the container name might be **xyz-information**. Substatements to the **output** statement define nodes under the RPC's **output** node. In the XML output, this would translate into XML elements under the **<rpc-reply>** element.

```
output {
   container output-container-name {
```

```
        ...
    }
}
```

Within the root container, you can include **leaf** and **container** statements. Leaf statements describe the data included in the RPC output for that container.

```
output {
   container output-container-name {
      container container-name {
         leaf output-param1 {
            type type;
            description description;
         }
         // additional leaf definitions
      }
   }
}
```

By default, the format for RPC output is XML. You can also define formatted ASCII output that is displayed when you execute the operational command for that RPC in the CLI or when you request the RPC output in text format.

NOTE: Starting in Junos OS Release 17.3, the CLI formatting for a custom RPC is defined within the junos-odl:format extension statement. In earlier releases, the CLI formatting is defined using a container that includes the junos-odl:cli-format statement.

- Starting in Junos OS Release 17.3, you define the CLI formatting by defining a **junos-odl:format** statement, which is a Junos OS ODL extension statement.

```
output {
   container output-container-name {
      container container-name {
         leaf output-param1 {
            type type;
            description description;
            // optional formatting statements
         }
         // additional leaf definitions
         junos-odl:format container-name-format {
            // CLI formatting for the parent container
         }
      }
      // Additional containers
   }
}
```

- Prior to Junos OS Release 17.3, you define the CLI formatting for a given container within a child container that includes the **junos-odl:cli-format** statement.

```
         container container-name-format {
            junos-odl:cli-format;
```

```
                    // CLI formatting for the parent container
                }
```

Within the statement or container that defines the CLI formatting, you can customize the RPC's CLI output by using statements defined in the Junos OS ODL extensions module. For more information about rendering formatted ASCII output, see "Customizing YANG RPC Output on Devices Running Junos OS" on page 348. You can also stipulate when the data in a particular container is emitted in an RPC's CLI output. For information about constructing different levels of output for the same RPC, see "Defining Different Levels of Output in Custom YANG RPCs for Devices Running Junos OS" on page 365.

To use the RPC on a device running Junos OS, download the module and action script to the device, and add the files to a new or existing YANG package by issuing the **request system yang add** or **request system yang update** operational command. To execute the RPC in the CLI, issue the command defined by the **junos:command** statement. To execute the RPC remotely, use the RPC name in an RPC request operation.

NOTE: Starting in Junos OS Release 17.3R1, when you load custom YANG data models onto the device, you do not need to explicitly load any required Junos OS extension modules. In earlier releases, you must load the Junos OS extension modules for any packages that use the modules.

When you execute the RPC in the CLI by issuing the command defined by the **junos:command** statement, the device displays the RPC output in the CLI format defined by the RPC. If the RPC does not define CLI formatting, by default, no output is displayed for that RPC in the CLI. However, you can still display the XML output for that RPC in the CLI by appending the **| display xml** filter to the command.

For more information about YANG RPCs, see RFC 6020, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*, and related RFCs.

Release History Table

Release	Description
19.2R1	Starting in Junos OS Release 19.2R1, custom YANG RPCs support input parameters of type **empty** when executing the RPC's command in the Junos OS CLI.
17.3R1	Starting in Junos OS Release 17.3, the **action-execute** statement is a substatement to **command**.
17.3R1	Starting in Junos OS Release 17.3, the CLI formatting for a custom RPC is defined within the **junos-odl:format** extension statement.
17.3R1	Starting in Junos OS Release 17.3R1, when you load custom YANG data models onto the device, you do not need to explicitly load any required Junos OS extension modules.

Related Documentation

- Creating Action Scripts for YANG RPCs on Devices Running Junos OS on page 324
- Using Custom YANG RPCs on Devices Running Junos OS on page 332
- Example: Using a Custom YANG RPC to Retrieve Operational Information on Devices Running Junos OS on page 334
- Understanding Junos OS YANG Extensions for Formatting RPC Output on page 345
- Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS on page 299

Creating Action Scripts for YANG RPCs on Devices Running Junos OS

You can add YANG data models that define custom remote procedure calls (RPCs) on supported devices running Junos OS. When you add a nonnative YANG RPC to a device, you must also supply an action script that serves as the RPC's handler. Action scripts are Python or Stylesheet Language Alternative SyntaX (SLAX) scripts that retrieve the information and perform the operations required by the RPC and return any necessary XML output elements as defined in the RPC **output** statement. The RPC definition references the action script, which is invoked when the RPC is executed.

Action scripts can be written in SLAX or Python. SLAX action scripts are similar to SLAX op scripts and can perform any function available through the RPCs supported by the Junos XML management protocol and the Junos XML API. Python action scripts can leverage all of the features and constructs in the Python language, which provides increased flexibility over SLAX scripts. In addition, Python action scripts support Junos PyEZ APIs, which facilitate executing RPCs and performing operational and configuration tasks on devices running Junos OS. Python scripts can also leverage the **lxml** library, which simplifies XPath handling.

This topic discusses how to create an action script, including how to parse the RPC input arguments, emit the XML output, access operational and configuration data in the script, and how to validate and load the script on a device.

Action Script Boilerplate

SLAX action scripts must include the necessary boilerplate for both basic script functionality as well as any optional functionality used within the script such as the Junos OS *extension functions* and *named templates*. In addition, the script must declare all RPC input parameters using the **param** statement. The SLAX action script boilerplate is as follows:

```
version 1.0;

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

import "/var/db/scripts/import/junos.xsl";

param $input-param1;
param $input-param2;

match / {
    <action-script-results> {
        /* insert your code here */
    }
}
```

Python action scripts must include an interpreter directive line and import any objects that are used by the script. Python action scripts that include the **import jcs** statement can use Junos OS *extension functions* and supported *named template* functionality in the script. If the script uses Junos PyEZ, it must import the necessary classes from the **jnpr.junos** module.

```
#!/usr/bin/env python
from jnpr.junos import Device
import jcs
```

Parsing RPC Input Arguments

An RPC can define input parameters using the optional **input** statement. When you execute an RPC and provide input arguments, Junos OS invokes the RPC's action script and passes those arguments to the script. In a Python or SLAX action script, you can

access the RPC input arguments in the same manner as you would access command-line arguments for a normal Python script or a Junos OS SLAX op script, respectively.

Consider the following **input** statement for the **get-host-status** RPC:

```
rpc get-host-status {
   description "RPC example to retrieve host status";

   junos:command "show host-status" {
      junos:action-execute {
         junos:script "rpc-host-status.py";
      }
   }

   input {
     leaf hostip {
       description "IP address of the target host";
       type string;
     }
     leaf level {
       type enumeration {
         enum brief {
           description "Display brief output";
         }
         enum detail {
           description "Display detailed output";
         }
       }
     }
     leaf test {
       description "empty argument";
       type empty;
     }
   }
   ...
```

The RPC can be executed in the CLI or through a NETCONF or Junos XML protocol session. For example:

```
user@host> show host-status hostip 198.51.100.1 level detail test
```

```
<rpc>
  <get-host-status>
    <hostip>198.51.100.1</hostip>
    <level>detail</level>
    <test/>
  </get-host-status>
</rpc>
```

NOTE: Starting in Junos OS Release 19.2R1, custom YANG RPCs support input parameters of type **empty** when executing the RPC's command in the Junos OS CLI, and the value passed to the action script is the parameter name. In earlier releases, input parameters of type **empty** are only supported when executing the RPC in a NETCONF or Junos XML protocol session, and the value passed to the action script is the string **'none'**.

In SLAX action scripts, you must declare input parameters using the **param** statement. The script assigns the value for each passed argument to the corresponding parameter, which can then be referenced throughout the script. If a parameter is type **empty**, the parameter name is passed in as its value. You must include the dollar sign ($) symbol both when you declare the parameter and when you access its value.

```
param $hostip;
param $level;
param $test
```

NOTE: For more information about SLAX parameters, see *SLAX Parameters Overview* in the *Automation Scripting Feature Guide*.

For Python action scripts:

- The first argument passed to the script is always the action script file path.
- The next arguments in the list are the name and value for each user-supplied input parameter.

  NOTE: When you execute the RPC's command in the CLI, the arguments are passed to the script in the order given on the command line. In a NETCONF or Junos XML protocol session, the order of arguments in the XML is arbitrary, so the arguments are passed to the script in the order that they are declared in the RPC **input** statement.

- The last two arguments in the list, which are supplied by the system and not the user, are **'rpc_name'** and the name of the RPC.

In a Python script, you can access the input arguments through the **sys.argv** list. For the previous example, the **sys.argv** input argument list is:

```
['/var/db/scripts/action/rpc-host-status.py', 'hostip', '198.51.100.1', 'level',
 'detail', 'test', 'test', 'rpc_name', 'get-host-status']
```

The following sample Python code demonstrates one way to extract the value for each argument from the **sys.arv** list for the example RPC. The example first defines a dictionary containing the possible argument names as keys and a default value for each argument. The code then checks for each key in the **sys.argv** argument list and retrieves the index of the argument name in the list, if present. The code then extracts the argument's value at the adjacent index position, and stores it in the dictionary for the appropriate key. This method ensures that if the arguments are passed to the script in a different order during execution, the correct values are retrieved.

```
import sys

# Define default values for arguments
args = {'hostip': None, 'level': 'brief', 'test': None}

# Retrieve user input and store the values in the args dictionary
for arg in args.keys():
    if arg in sys.argv:
        index = sys.argv.index(arg)
        args[arg] = sys.argv[index+1]
```

Retrieving Operational and Configuration Data

Action scripts can retrieve operational and configuration data from a device running Junos OS and then parse the data for necessary information. SLAX action scripts can retrieve information from the device by executing RPCs supported by the Junos XML management protocol and the Junos XML API. Python action scripts can retrieve operational and configuration information by using Junos PyEZ APIs or by using the **cli -c '*command*'** to execute CLI commands in the action script as you would from the shell. To retrieve operational information with the **cli -c** method, include the desired operational command. To retrieve configuration information, use the **show configuration** command.

The following SLAX snippet executes the **show interfaces** command on the local device by using the equivalent **<get-interface-information>** request tag:

```
var $rpc = <get-interface-information>;
var $out = jcs:invoke($rpc);
/* parse for relevant information and return as XML tree for RPC output */
```

The following Python code uses Junos PyEZ to execute the **show interfaces** command by using the equivalent Junos PyEZ **get_interface_information** RPC:

```
from jnpr.junos import Device
from lxml import etree

dev = Device()
dev.open()
res = dev.rpc.get_interface_information()
dev.close()
 # parse for relevant information and return as XML tree for RPC output
```

NOTE: For information about using Junos PyEZ to execute RPCs on devices running Junos OS, see Using Junos PyEZ to Execute RPCs on Devices Running Junos OS.

The following Python code executes the **show interfaces | display xml** command and converts the string output into an XML tree that can be parsed for the required data using XPath constructs:

```
import subprocess
from lxml import etree

cmd = ['cli', '-c', 'show interfaces | display xml']
proc = subprocess.Popen(cmd, stdout=subprocess.PIPE)
tmp = proc.stdout.read()
root = etree.fromstring(tmp.strip())
# parse for relevant information and return as XML tree for RPC output
```

Emitting the RPC XML Output

An RPC can define output elements using the optional **output** statement. The action script must define and emit any necessary XML elements for the RPC output. The XML hierarchy emitted by the script should reflect the tree defined by the containers and leaf statements in the definition of the RPC **output** statement. To return the XML output, the action script must emit the RPC output hierarchy, and only the output hierarchy. SLAX scripts must use the **copy-of** statement to emit the XML, and Python scripts can use **print** statements.

For example, consider the following YANG RPC **output** statement:

```
      output {
        container host-status-information {
          container host-status-info {
            leaf host {
              type string;
              description "Host IP";
            }
            leaf status {
              type string;
              description "Host status";
            }
            leaf date {
              type string;
              description "Date and time";
            }
          }
        }
      }
```

The action script must generate and emit the corresponding XML output, for example:

```
<host-status-information>
  <host-status-info>
    <host>198.51.100.1</host>
    <status>Active</status>
    <date>2016-10-10</date>
  </host-status-info>
  <host-status-info>
    <host>198.51.100.2</host>
    <status>Inactive</status>
    <date>2016-10-10</date>
```

```
  </host-status-info>
</host-status-information>
```

After retrieving the values for the required output elements, a Python script might emit the XML output hierarchy by using the following code:

```
from lxml import etree
...

xml = '''
<host-status-information>
  <host-status-info>
    <host>{0}</host>
    <status>{1}</status>
    <date>{2}</date>
  </host-status-info>
</host-status-information>
'''.format(hostip, pingstatus, now)

tree = etree.fromstring(xml)
print (etree.tostring(tree))
```

Similarly, a SLAX action script might use the following:

```
var $node = {
    <host-status-information> {
        <host-status-info> {
            <host> $ip;
            <status> $pingstatus;
            <date> $date;
        }
    }
}
copy-of $node;
```

Validating and Loading Action Scripts on a Device

In your YANG RPC definition, you specify the RPC's action script by including the **junos:command** and **junos:action-execute** statements and the **junos:script** substatement, which takes the action script's filename as its value. You must define one and only one action script for each RPC. For example:

```
rpc rpc-name {
   ...
   junos:command "show sw-info" {
      junos:action-execute {
         junos:script "sw-info.py";
      }
   }
   ...
}
```

NOTE: Starting in Junos OS Release 17.3, the **action-execute** statement is a substatement to **command**. In earlier releases, the **action-execute** and **command** statements are placed at the same level, and the **command** statement is optional.

NOTE: YANG modules that define RPCs for devices running Junos OS must import the Junos OS DDL extensions module.

Python action scripts must meet the following requirements before you can execute the scripts on devices running Junos OS.

- File owner is either root or a user in the Junos OS **super-user** login class.
- Only the file owner has write permission for the file.
- Script includes an interpreter directive line such as **#!/usr/bin/env python**.
- The **language python** statement is configured at the **[edit system scripts]** hierarchy level.

NOTE: Users can only execute unsigned Python scripts on devices running Junos OS when the script's file permissions include read permission for the first class that the user falls within, in the order of user, group, or others.

You can validate the syntax of an action script in the Junos OS CLI by issuing the **request system yang validate action-script** command and providing the path to the script. For example:

```
user@host> request system yang validate action-script /var/tmp/sw-info.py
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
```

To use an action script, you must load it onto the device with the YANG module that contains the corresponding RPC. You use the **request system yang add** or **request system yang update** commands to add YANG modules and their associated action scripts to a new or existing YANG package on the device. After you add the modules and action scripts to the device, you can execute your custom RPCs. When you execute an RPC, the device invokes the referenced script.

Release History Table

Release	Description
19.2R1	Starting in Junos OS Release 19.2R1, custom YANG RPCs support input parameters of type **empty** when executing the RPC's command in the Junos OS CLI, and the value passed to the action script is the parameter name.
17.3R1	Starting in Junos OS Release 17.3, the **action-execute** statement is a substatement to **command**.

Related Documentation

- Creating Custom RPCs in YANG for Devices Running Junos OS on page 317
- Using Custom YANG RPCs on Devices Running Junos OS on page 332
- Example: Using a Custom YANG RPC to Retrieve Operational Information on Devices Running Junos OS on page 334

Using Custom YANG RPCs on Devices Running Junos OS

You can add YANG data models that define custom RPCs on supported devices running Junos OS. Creating custom RPCs enables you to precisely define the input parameters and operations and the output fields and formatting for your specific operational tasks on those devices.

To add an RPC to a device running Junos OS, download the YANG module that defines the RPC, along with any required action scripts to the device, and add the files to a new or existing YANG package by issuing the **request system yang add** or **request system yang update** operational command. For detailed information about adding YANG modules to devices running Junos OS, see "Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS" on page 299.

NOTE: Starting in Junos OS Release 17.3R1, when you load custom YANG data models onto the device, you do not need to explicitly load any required Junos OS extension modules. In earlier releases, you must load the Junos OS extension modules for any packages that use the modules.

After you add the modules and action scripts to the device, you can execute the RPC either locally, provided that the RPC definition includes the **junos:command** statement, or remotely. To execute an RPC in the Junos OS CLI, issue the command defined by the RPC's **junos:command** statement. To execute an RPC remotely, use the RPC name in an RPC request operation.

Consider the following YANG module and RPC definition:

```
module sw-info {
  namespace "http://yang.juniper.net/examples/rpc-cli";
  prefix rpc-cli;
```

```
  import junos-extension {
    prefix junos;
  }

  rpc get-sw-info {
    description "Show software information";
    junos:command "show sw-info" {
       junos:action-execute {
         junos:script "sw-info.py";
      }
    }
    input {
      leaf routing-engine {
        type string;
        description "Routing engine for which to display information";
      }
      ...
    }
    output {
      ...
    }
  }
}
```

NOTE: Starting in Junos OS Release 17.3, the **action-execute** statement is a substatement to **command**. In earlier releases, the **action-execute** and **command** statements are placed at the same level, and the **command** statement is optional.

To execute this RPC in the Junos OS CLI, issue the **show sw-info** command defined by the **junos:command** statement, and include any required or optional input parameters. For example:

```
user@host> show sw-info routing-engine re0
```

To execute this RPC remotely, send an RPC request that uses the RPC name for the request tag, and include any required or optional input parameters.

```
  <rpc>
     <get-sw-info>
       <routing-engine>re0</routing-engine>
     </get-sw-info>
  </rpc>
```

When you execute a custom RPC, the device invokes the action script that is defined in the **junos:script** statement, which in this example is the **sw-info.py** script. An RPC's action script should emit any necessary XML elements for that RPC's output.

When you execute an RPC in the Junos OS CLI by issuing the command defined by the **junos:command** statement, the device displays the RPC output, if there is any, using the CLI formatting defined by the RPC. If the RPC does not define CLI formatting, the device

does not display any output for that RPC in the CLI. However, you can still display the RPC's XML output in the CLI by appending **| display xml** to the command.

```
user@host> show sw-info routing-engine re0 | display xml
```

When you execute an RPC remotely, the RPC output defaults to XML. However, you can specify a different output format by including the **format** attribute in the opening request tag of the RPC. To display CLI formatting, provided that the RPC defines this format, set the **format** attribute to **text** or **ascii**. To display the output in JavaScript Object Notation (JSON), set the **format** attribute to **json**. For example:

```
<rpc>
   <get-sw-info format="text">
     <routing-engine>re0</routing-engine>
   </get-sw-info>
</rpc>
```

Release History Table

Release	Description
17.3R1	Starting in Junos OS Release 17.3R1, when you load custom YANG data models onto the device, you do not need to explicitly load any required Junos OS extension modules.

Related Documentation

- Creating Custom RPCs in YANG for Devices Running Junos OS on page 317
- Creating Action Scripts for YANG RPCs on Devices Running Junos OS on page 324
- Example: Using a Custom YANG RPC to Retrieve Operational Information on Devices Running Junos OS on page 334
- Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS on page 299

Example: Using a Custom YANG RPC to Retrieve Operational Information on Devices Running Junos OS

You can add YANG data models that define custom RPCs on devices running Junos OS. Creating custom RPCs enables you to precisely define the input parameters and operations and the output fields and formatting for your specific operational tasks on those devices. This example presents a custom RPC and action script that retrieve operational information from the device and display customized CLI output.

The RPC is added to the Junos OS schema on the device. When the RPC is executed in the CLI, it prints the name and operational status for the requested physical interfaces.

- Requirements on page 335
- Overview of the RPC and Action Script on page 335
- Loading the RPC on the Device on page 341
- Enabling Execution of Python Scripts on page 342

Requirements

This example uses the following hardware and software components:

- MX Series router running Junos OS Release 17.3R1.

Overview of the RPC and Action Script

The YANG module presented in this section defines a custom RPC to return the name and operational status of certain physical interfaces. The YANG module **rpc-interface-status** is saved in the **rpc-interface-status.yang** file. The module imports the Junos OS extension modules, which provide the extensions required to execute custom RPCs on the device and to customize the CLI output.

The module defines the **get-interface-status** RPC. The **<get-interface-status>** request tag is used to remotely execute the RPC on the device. In the RPC definition, the **junos:command** statement defines the command that is used to execute the RPC in the CLI, which in this case is **show intf status**.

The **junos:action-execute** and **junos:script** statements define the action script that is invoked when the RPC is executed. This example uses a Python action script named **rpc-interface-status.py** to retrieve the information required by the RPC and return the XML output elements as defined in the RPC **output** statement.

```
rpc get-interface-status {
   description "RPC example to retrieve interface status";

   junos:command "show intf status" {
      junos:action-execute {
        junos:script "rpc-interface-status.py";
      }
   }
  ...
```

NOTE: Starting in Junos OS Release 17.3, the action-execute statement is a substatement to command. In earlier releases, the action-execute and command statements are placed at the same level, and the command statement is optional.

The RPC has one input parameter named **match**, which determines the interfaces to include in the output. When the user executes the RPC, the user includes a string that matches on the desired interfaces, for example ge-0*. An empty string ("") matches on all interfaces. The action script defines the default value for **match** as an empty string, so if the user omits this argument, the output will include information for all interfaces.

```
input {
  leaf match {
    description "Requested interface match condition";
```

```
          type string;
        }
      }
```

The RPC also defines the output nodes that must be emitted by the corresponding action script. The root node is the **<interface-status-info>** element, which contains zero or more **<status-info>** elements that enclose the **<interface>** and **<status>** nodes for a matched interface. The **junos-odl:format** statement **interface-status-info-format** defines the formatting for the output that is displayed in the CLI. This node is not emitted in the output XML tree.

```
      output {
        container interface-status-info {
          list status-info {
            leaf interface {
              type string;
              description "Physical inteface name";
            }
            leaf status {
              type string;
              description "Operational status";
            }
            junos-odl:format interface-status-info-format {
              ...
            }
          }
        }
      }
```

This example presents two versions of the Python action script. The scripts demonstrate different means to retrieve the operational command output, but both scripts emit identical RPC output. The first action script uses the Python **subprocess** module to execute the **show interfaces *match-value* | display xml** command and then converts the string output into XML. The second action script uses Junos PyEZ to execute the RPC equivalent of the **show interfaces *match-value*** command. Both scripts use identical code to parse the command output and extract the name and operational status for each physical interface. The scripts construct the XML for the RPC output and then print the output, which returns the information back to the device. The XML tree must exactly match the hierarchy defined in the RPC.

NOTE: Devices running Junos OS define release-dependent namespaces for many of the elements in the operational output, including the **<interface-information>** element. In order to make the RPC Junos OS-release independent, the code uses the **local-name()** function in the XPath expressions for these elements. You might choose to include the namespace mapping as an argument to **xpath()** and qualify the elements with the appropriate namespace.

The module containing the RPC and the action script file are added to the device as part of a new YANG package named **intf-rpc**.

YANG Module The YANG module, **rpc-interface-status.yang**, defines the RPC, the command used to execute the RPC in the CLI, and the name of the action script to invoke when the RPC is executed. The base name of the file must match the module name.

```
/*
* Copyright (c) 2014 Juniper Networks, Inc.
* All rights reserved.
*/

module rpc-interface-status {
  namespace "http://yang.juniper.net/examples/rpc-cli";
  prefix rpc-cli;

  import junos-extension-odl {
    prefix junos-odl;
  }
  import junos-extension {
    prefix junos;
  }

  organization
    "Juniper Networks, Inc.";

  description
    "Junos OS YANG module for RPC example";

  rpc get-interface-status {
     description "RPC example to retrieve interface status";

     junos:command "show intf status" {
        junos:action-execute {
          junos:script "rpc-interface-status.py";
        }
     }

     input {
       leaf match {
         description "Requested interface match condition";
         type string;
       }
     }
     output {
       container interface-status-info {
         list status-info {
           leaf interface {
             type string;
             description "Physical interface name";
           }
           leaf status {
             type string;
             description "Operational status";
           }
           junos-odl:format interface-status-info-format {
             junos-odl:header "Physical Interface - Status\n";
             junos-odl:indent 5;
             junos-odl:comma;
             junos-odl:space;
             junos-odl:line {
               junos-odl:field "interface";
               junos-odl:field "status";
```

```
            }
          }
        }
      }
    }
  }
}
```

The corresponding action script is **rpc-interface-status.py**. This example presents two action scripts that use different means to retrieve the data. Both scripts emit the same RPC XML output.

Action Script The following action script uses the Python **subprocess** module to execute the operational command and retrieve the data:

```
#!/usr/bin/python
import sys
import subprocess
from lxml import etree

def get_device_info(cmd):
    """
    Execute Junos OS operational command and parse output
    :param: str cmd: operational command to execute
    :returns: List containing the XML data for each interface
    """

    # execute Junos OS operational command and retrieve output
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE)
    tmp = proc.stdout.read()
    root = etree.fromstring(tmp.strip())

    xml_items = []

    # parse output for required data
    for intf in root.xpath("/rpc-reply \
        /*[local-name()='interface-information'] \
        /*[local-name()='physical-interface']"):

        # retrieve data for the interface name and operational status
        name = intf.xpath("*[local-name()='name']")[0].text
        oper_status = intf.xpath("*[local-name()='oper-status']")[0].text

        # append the XML for each interface to a list
        xml_item = etree.Element('status-info')
        interface = etree.SubElement(xml_item, 'interface')
        interface.text = name
        status = etree.SubElement(xml_item, 'status')
        status.text = oper_status
        xml_items.append(xml_item)

    return xml_items


def generate_xml(cmd):
    """
    Generate the XML tree for the RPC output
    :param: str cmd: operational command from which to retrieve data
```



```
    :returns: XML tree for the RPC output
    """

    xml = etree.Element('interface-status-info')

    intf_list_xml = get_device_info(cmd)
    for intf in intf_list_xml:
        xml.append(intf)
    return xml


def main():

    args = {'match': ""}

    for arg in args.keys():
        if arg in sys.argv:
            index = sys.argv.index(arg)
            args[arg] = sys.argv[index+1]

    # define the operational command from which to retrieve information
    cli_command = 'show interfaces ' + args['match'] + ' | display xml'
    cmd = ['cli', '-c', cli_command]

    # generate the XML for the RPC output
    rpc_output_xml = generate_xml(cmd)

    # print RPC output
    print (etree.tostring(rpc_output_xml, pretty_print=True))


if __name__ == '__main__':

    main()
```

Action Script (Junos PyEZ)

The following action script uses Junos PyEZ to execute the operational command and retrieve the data:

```
#!/usr/bin/python
import sys
from jnpr.junos import Device
from jnpr.junos.exception import *
from lxml import etree

def get_device_info(match):
    """
    Execute Junos OS operational command and parse output
    :param: str match: interface match condition
    :returns: List containing the XML data for each interface
    """

    # execute Junos OS operational command and retrieve output
    dev = Device()
    dev.open()
    if (match == ""):
        root=dev.rpc.get_interface_information()
    else:
        root = dev.rpc.get_interface_information(interface_name=match)
```

```
    dev.close()

    xml_items = []

    # parse output for required data
    for intf in root.xpath("/rpc-reply \
        /*[local-name()='interface-information'] \
        /*[local-name()='physical-interface']"):

        # retrieve data for the interface name and operational status
        name = intf.xpath("*[local-name()='name']")[0].text
        oper_status = intf.xpath("*[local-name()='oper-status']")[0].text

        # append the XML for each interface to a list
        xml_item = etree.Element('status-info')
        interface = etree.SubElement(xml_item, 'interface')
        interface.text = name
        status = etree.SubElement(xml_item, 'status')
        status.text = oper_status
        xml_items.append(xml_item)

    return xml_items


def generate_xml(match):
    """
    Generate the XML tree for the RPC output
    :param: str match: interface match condition
    :returns: XML tree for the RPC output
    """

    xml = etree.Element('interface-status-info')

    intf_list_xml = get_device_info(match)
    for intf in intf_list_xml:
        xml.append(intf)
    return xml


def main():

    args = {'match': ""}

    for arg in args.keys():
        if arg in sys.argv:
            index = sys.argv.index(arg)
            args[arg] = sys.argv[index+1]

    # generate the XML for the RPC output
    rpc_output_xml = generate_xml(args['match'])

    # print RPC output
    print (etree.tostring(rpc_output_xml, pretty_print=True))

if __name__ == '__main__':

    main()
```

Loading the RPC on the Device

To add the RPC and action script to the Junos OS schema on a device running Junos OS:

1. Download the YANG module and action script to the device running Junos OS.

2. Ensure that the Python action script meets the following requirements:
   - File owner is either root or a user in the Junos OS **super-user** login class.
   - Only the file owner has write permission for the file.

3. (Optional) Validate the syntax for the YANG module and action script.

   ```
   user@host> request system yang validate module /var/tmp/rpc-interface-status.yang
   action-script /var/tmp/rpc-interface-status.py
   YANG modules validation : START
   YANG modules validation : SUCCESS
   Scripts syntax validation : START
   Scripts syntax validation : SUCCESS
   ```

4. Add the YANG module and action script to a new YANG package.

   ```
   user@host> request system yang add package intf-rpc module
   /var/tmp/rpc-interface-status.yang action-script /var/tmp/rpc-interface-status.py
   YANG modules validation : START
   YANG modules validation : SUCCESS
   Scripts syntax validation : START
   Scripts syntax validation : SUCCESS
   TLV generation: START
   TLV generation: SUCCESS
   Building schema and reloading /config/juniper.conf.gz ...
   Activating /config/juniper.conf.gz ...
   mgd: commit complete
   Restarting mgd ...
   ```

   NOTE: Starting in Junos OS Release 17.3R1, when you load custom YANG data models onto the device, you do not need to explicitly load any required Junos OS extension modules. In earlier releases, you must load the Junos OS extension modules for any packages that use the modules.

5. When the system prompts you to restart the Junos OS CLI, press **Enter** to accept the default value of **yes**, or type **yes** and press **Enter**.

   ```
   WARNING: cli has been replaced by an updated version:
   ...
   Restart cli using the new version ? [yes,no] (yes) yes

   Restarting cli ...
   ```

Enabling Execution of Python Scripts

To enable the device to execute unsigned Python scripts, configure the **language python** statement.

```
[edit]
user@host# set system scripts language python
user@host# commit and-quit
```

Verifying the RPC

Purpose Verify that the RPC is working as expected.

Action From operational mode, execute the RPC in the CLI by issuing the command defined by the **junos:command** statement in the RPC definition, and include the **match** input argument. In this example, the match argument is used to match on all interfaces that start with ge-0.

```
user@host> show intf status match ge-0*
Physical Interface - Status
     ge-0/0/0, up
     ge-0/0/1, up
     ge-0/0/2, up
     ge-0/0/3, up
     ge-0/0/4, up
     ge-0/0/5, up
     ge-0/0/6, up
     ge-0/0/7, up
     ge-0/0/8, up
     ge-0/0/9, up
     ge-0/1/0, up
     ge-0/1/1, up
     ge-0/1/2, up
     ge-0/1/3, up
     ge-0/1/4, up
     ge-0/1/5, up
     ge-0/1/6, up
     ge-0/1/7, up
     ge-0/1/8, up
     ge-0/1/9, up
```

You can also adjust the match condition to return different sets of interfaces. For example:

```
user@host> show intf status match *e-0/*/0
Physical Interface - Status
     ge-0/0/0, up
     pfe-0/0/0, up
     ge-0/1/0, up
     xe-0/2/0, up
     xe-0/3/0, up
```

To return the same output in XML format, append the **| display xml** filter to the command.

```
user@host> show intf status match *e-0/*/0 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.3R1/junos">
    <interface-status-info>
          <status-info>
            <interface>ge-0/0/0</interface>
            <status>up</status>
          </status-info>
          <status-info>
            <interface>pfe-0/0/0</interface>
            <status>up</status>
          </status-info>
          <status-info>
            <interface>ge-0/1/0</interface>
            <status>up</status>
          </status-info>
```



```
        <status-info>
          <interface>xe-0/2/0</interface>
          <status>up</status>
        </status-info>
        <status-info>
          <interface>xe-0/3/0</interface>
          <status>up</status>
        </status-info>
    </interface-status-info>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

NOTE: To match on all interfaces, either omit the match argument or set the value of the argument to an empty string ("").

Meaning When the RPC is executed, the action script is invoked. The action script runs the operational command to retrieve interface information from the device, parses the output for the desired information, and prints the XML hierarchy for the RPC output as defined in the RPC **output** statement. When the RPC is executed in the CLI, the device uses the CLI formatting defined in the RPC to convert the XML output into the displayed CLI output. To return the original XML output, append the **| display xml** filter to the command.

NOTE: When the RPC is executed remotely using the RPC request tag, the default format for the output is XML.

Troubleshooting RPC Execution Errors

Problem **Description:**

When you execute the RPC, the device generates the following error:

```
error: open failed: /var/db/scripts/action/rpc-interface-status.py: Permission
denied
```

Cause The user who invoked the RPC does not have the necessary permissions to execute the corresponding Python action script.

Solution Users can only execute unsigned Python scripts on devices running Junos OS when the script's file permissions include read permission for the first class that the user falls within, in the order of user, group, or others.

Verify whether the script has the necessary permissions for that user to execute the script, and adjust the permissions, if appropriate. If you update the permissions, you must also update the YANG package in order for this change to take effect. For example:

```
% ls -l rpc-interface-status.py
-rw-------  1 root    wheel  2215 Apr 20 11:36 rpc-interface-status.py

% chmod 644 rpc-interface-status.py
% ls -l rpc-interface-status.py
-rw-r--r--  1 root    wheel  2215 Apr 20 11:36 rpc-interface-status.py

% cli

user@host> request system yang update intf-rpc action-script /var/tmp/rpc-interface-status.py
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
```

Related Documentation

- Creating Custom RPCs in YANG for Devices Running Junos OS on page 317
- Creating Action Scripts for YANG RPCs on Devices Running Junos OS on page 324
- Using Custom YANG RPCs on Devices Running Junos OS on page 332
- Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS on page 299

Understanding Junos OS YANG Extensions for Formatting RPC Output

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

The Junos OS Output Definition Language (ODL) defines the transformation of the XML-tagged data into the formatted ASCII output that is displayed when you execute a command in the CLI or request RPC output in text format. The Junos OS ODL extensions module defines YANG extensions for the ODL, which you can include in custom YANG RPCs to translate the XML RPC reply into formatted ASCII output.

The YANG RPC **output** statement defines output parameters to the RPC operation. Within the RPC **output** statement, you can include ODL extension statements to customize the RPC's output. Table 13 on page 346 outlines the available statements, provides a brief description of each statement's formatting impact, and specifies the locations where the statement can be defined within the RPC **output** statement.

You include some ODL extension statements under the leaf statement that defines the data, and you include others within the output container or at various levels within the **format** statement, which defines the CLI formatting. The placement of a statement within the **format** statement determines the statement's scope, which might apply to a single field, all fields in a line, or all fields in all lines of output. Statements that can be defined at any level in the **format** statement can be included at the top level as a direct child of the **format** statement, directly under the **line** statement, or within a **field** statement.

NOTE: Starting in Junos OS Release 17.3, the CLI formatting for a custom RPC is defined within the junos-odl:format extension statement. In earlier releases, the CLI formatting is defined using a container that includes the junos-odl:cli-format statement.

Table 13: Statements in the Junos OS ODL Extensions Module

Statement	Description	Placement Within RPC output Statement
blank-line	Insert a blank line between each repetition of data when the RPC reply returns the same set of information for multiple entities.	**format** statement (top level)
capitalize	Capitalize the first word of a node's value in an output field.	**format** statement (any level)
cli-format	Indicate that the enclosing container defines the CLI formatting for the parent container. The formatting container is not included as a node in the XML RPC reply. This statement is obsolete starting in Junos OS Release 17.3. Use the **format** statement instead.	formatting container (top level)
colon	Insert a colon following the node's label in an output field. This statement is only used in conjunction with the **leading** statement to insert the formal name of the node, as defined by the **formal-name** statement, and a colon before the value of the node in the output field.	**format** statement (any level)
comma	Insert a comma after a node's value in an output field.	**format** statement (any level)
default-text	Specify the text to display when the node corresponding to an output field is missing.	**field** statement
explicit	Direct the renderer to display a value that is unrelated to the node name or its contents. This statement is used in Junos OS RPCs only and cannot be included in custom RPCs.	–
field	Map a leaf node in the output tree to a field in the formatted ASCII output.	**line** statement
fieldwrap	Wrap a field's complete contents to the following line when the current line is wider than the screen. Omitting this statement causes the output to wrap without regard for appropriate word breaks or the prevailing margin.	**field** statement

Table 13: Statements in the Junos OS ODL Extensions Module (continued)

Statement	Description	Placement Within RPC **output** Statement
float	Enable the value in a field to move to the left into an empty field. Use this statement to indicate subsequent mutually exclusive values for a set of adjacent fields so that only the leftmost field includes one of these possible values. If the leftmost field is not populated by the first value, a value mapped to a subsequent field that includes the **float** statement can move into the empty field.	**field** statement
formal-name	Define the label that precedes a node's value in an output field whenever the field for that node includes the **leading** statement in the formatting instructions.	**leaf** node
format	Define the CLI formatting for the parent container within the RPC **output** statement. Starting in Junos OS Release 17.3, the CLI formatting is defined within the **format** statement rather than within a container that includes the **cli-format** statement.	output container or as a substatement to the **style** statement.
header	Define a header row in the CLI output.	**format** statement (top level)
header-group	Require that only the first header string as defined by the **header** statement be emitted in the CLI output for that header group.	**format** statement (top level)
indent	Indent all lines other than the header row by the specified number of spaces in the CLI output.	**format** statement (top level)
leading	Insert a label, which is defined by the **formal-name** statement in the definition of a leaf node, before the node's value in an output field.	**format** statement (any level)
line	Define the group of fields that comprises a single line of output.	**format** statement (top level)
no-line-break	Display multiple values on the same line in the case where multiple entities with the same tag names are emitted.	**format** statement (top level)
picture	Graphically specify the placement, justification, and width of the columns in a table in the RPC's formatted ASCII output.	**format** statement (top level)
space	Insert a space after the node's value in an output field. If the **space** statement is used in conjunction with the **comma** statement, the output inserts a comma and then a space after the node's value, in that order.	**format** statement (any level)
style	Define a format, or style, for the RPC output. Use this statement in conjunction with an enumerated input parameter that defines the names for each style. Define this statement with the appropriate style name to specify the CLI formatting for that style.	output container

Table 13: Statements in the Junos OS ODL Extensions Module (continued)

Statement	Description	Placement Within RPC output Statement
template	Explicitly define the format for an output field, including the output string and the placement of the node's value within that string. Use **%s** or **%d** to indicate the placement of the node's string or integer value, respectively, within the output string. If a leaf statement defines both a **template** and a **formal-name** statement, and the corresponding field's formatting instructions include the **leading** statement, the output displays the text defined for the **formal-name** statement and not the text defined for the **template** statement.	**leaf** node
truncate	Truncate a node's value to fit the field width defined by the **picture** statement if the node's contents would otherwise exceed the width of the field.	**field** statement
wordwrap	Wrap some of the field to the following line when the current line is wider than the screen. This statement should only be used for fields in the rightmost column of a table.	**field** statement

For more information about the structure of YANG RPCs, see "Creating Custom RPCs in YANG for Devices Running Junos OS" on page 317.

Release History Table

Release	Description
17.3R1	Starting in Junos OS Release 17.3, the CLI formatting for a custom RPC is defined within the **junos-odl:format** extension statement. In earlier releases, the CLI formatting is defined using a container that includes the **junos-odl:cli-format** statement.

Related Documentation

- Customizing YANG RPC Output on Devices Running Junos OS on page 348
- Creating Custom RPCs in YANG for Devices Running Junos OS on page 317
- Defining Different Levels of Output in Custom YANG RPCs for Devices Running Junos OS on page 365

Customizing YANG RPC Output on Devices Running Junos OS

You can create custom RPCs in YANG for devices running Junos OS. This enables you to precisely define the input parameters and operations and the output fields and formatting for specific operational tasks on devices running Junos OS.

When you execute an RPC on a device running Junos OS, it returns the RPC reply as an XML document. The Junos OS Output Definition Language (ODL) defines the transformation of the XML data into the formatted ASCII output that is displayed when you execute a command in the CLI or request RPC output in text format. The Junos OS ODL extensions module defines YANG extensions for the Junos OS ODL, which you can include in custom RPCs to specify the CLI formatting for the output. For a summary of

all the statements and their placement within the RPC **output** statement, see "Understanding Junos OS YANG Extensions for Formatting RPC Output" on page 345.

NOTE: Starting in Junos OS Release 17.3, the CLI formatting for a custom RPC is defined within the junos-odl:format extension statement. In earlier releases, the CLI formatting is defined using a container that includes the junos-odl:cli-format statement.

The following sections outline how to use the Junos OS ODL extension statements. Closely related statements are presented in the same section, and in some instances, a statement might be included in more than one section. The examples assume that the enclosing YANG module imports the Junos OS ODL extensions module and binds it to the **junos-odl** prefix. The examples use the **format** statement, which is introduced in Junos OS Release 17.3, to define the CLI formatting.

blank-line

The **blank-line** statement inserts a line between each repetition of data when the RPC reply returns the same set of information for multiple entities. For example, if the RPC reply returns data for multiple interfaces, the formatted ASCII output inserts a blank line between each interface's set of data.

```
Physical interface: so-1/1/0, Enabled, Physical link is Down
Interface index: 11, SNMP ifIndex: 41
...
```



```
Active defects : LOL, LOF, LOS, SEF, AIS-L, AIS-P

Physical interface: so-1/1/1, Enabled, Physical link is Down
Interface index: 12, SNMP ifIndex: 42
...
Active defects : LOL, LOF, LOS, SEF, AIS-L, AIS-P
```

To insert a blank line between each entity's data set, include the **blank-line** statement directly under the **format** statement.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         // leaf definitions
         junos-odl:format xyz-information-format {
            junos-odl:blank-line;
            // CLI formatting
          }
      }
   }
}
```

capitalize

The **capitalize** statement capitalizes the first word of a node's value in an output field. It does not affect the capitalization of a node's formal name. For example, if the RPC output includes a **state** node with the value **online**, the **capitalize** statement causes the value to be capitalized in the output.

```
State: Online
```

To capitalize the first word of the node's value, include the **capitalize** statement within the **format** statement. The placement of the statement determines the statement's scope and whether it affects a single field, all fields in a single line, or all lines.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf state {
            junos-odl:formal-name "State";
            type string;
            description "Interface state";
         }
         junos-odl:format xyz-information-format {
            junos-odl:header "xyz information\n";
            junos-odl:line {
               junos-odl:field "state" {
                  junos-odl:leading;
                  junos-odl:colon;
                  junos-odl:capitalize;
               }
            }
         }
      }
```

```
    }
}
```

cli-format

When you execute an RPC on a device running Junos OS, it returns the RPC reply as an XML document. Container and leaf nodes under the RPC **output** statement translate into XML elements in the RPC reply. In YANG RPCs for devices running Junos OS, you can also define custom formatted ASCII output that is displayed when you execute the RPC on the Junos OS command-line interface (CLI) or request RPC output in text format.

In Junos OS Release 17.2 and earlier releases, to create custom command output for a specific RPC output container, create a child container that includes the **cli-format** statement. The **cli-format** statement indicates that the enclosing container defines the CLI formatting for the parent container, and that this container should not be included as a node in the XML data of the RPC reply. Within the formatting container, map the data for the parent container to output fields, and use statements from the Junos OS ODL extensions module to specify how to display the output for that parent container.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         // leaf definitions
         container xyz-information-format {
            junos-odl:cli-format;
            // CLI formatting for the parent container
         }
      }
   }
}
```

To create custom command output for a specific RPC output container in Junos OS Release 17.3 and later releases, see "format" on page 358.

colon, formal-name, and leading

A node's formal name, or label, is the text that precedes a node's contents in the output when the **leading** statement is included in the formatting instructions for that node's output field. To create a label for a node, you must include the **formal-name** statement in the definition of the leaf node. In the following example, the **version** node has the formal name **Version**:

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            junos-odl:formal-name "Version";
            type string;
            description "Version";
         }
         ...
      }
```

```
    }
}
```

The **colon** statement inserts a colon after the node's label in an output field. If the formatting instructions include both the **colon** and **leading** statements, the node's label and a colon are inserted before the node's value in the output. For example:

```
Version: value
```

To insert the label and a colon in the output field, include the **leading** and **colon** statements within the **format** statement. The placement of the statements determines the scope and whether the statements affect a single field, all fields in a single line, or all lines.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            junos-odl:formal-name "Version";
            type string;
            description "Version";
         }
         junos-odl:format xyz-information-format {
            junos-odl:line {
               junos-odl:field "version" {
                   junos-odl:colon;
                   junos-odl:leading;
               }
            }
         }
      }
   }
}
```

When you execute the RPC, the label and a colon are included in the output for that field.

```
Version: value
```

comma

The **comma** statement appends a comma to the node's value in the output. It is used in conjunction with the **space** statement to create comma-delimited fields in a line of output. For example:

```
value1, Label2: value2, value3
```

To generate a comma and a space after a node's value in the output field, include the **comma** and **space** statements within the **format** statement. The placement of the statements within the **format** statement determines the scope. Placing the statements within a single field generates a comma and space for that field only. Placing the statements directly under the **format** statement applies the formatting to all fields.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            type string;
            description "Version";
         }
         // additional leaf definitions
         junos-odl:format xyz-information-format {
            junos-odl:comma;
            junos-odl:space;
            junos-odl:line {
               junos-odl:field "version";
               // additional fields
            }
         }
      }
   }
}
```

If you omit the **comma** statement in the formatting instructions in this example, the fields are separated using only a space. Junos OS automatically omits the comma and space after the last field in a line of output.

default-text

The **default-text** statement specifies the text to display when the node corresponding to an output field is missing.

To define the string to display in the formatted ASCII output when the node mapped to a field is missing, include the **default-text** statement and string within the **field** statement for that node.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf my-model {
            type string;
            description "Model";
         }
         junos-odl:format xyz-information-format {
            junos-odl:line {
               junos-odl:field "my-model" {
                  junos-odl:default-text "Model number not available.";
               }
            }
         }
      }
   }
}
```

When the node is missing in the RPC reply, the CLI output displays the default text.

```
Model number not available.
```

NOTE: The device only displays the default text when the node is missing. It does not display the text for nodes that are present but empty.

explicit

The **explicit** statement is only used in Junos OS RPCs and cannot be included in custom RPCs.

field and line

The **line** and **field** statements define lines in the RPC's formatted ASCII output and the fields within those lines. These statements can also be used with the **picture** statement to create a more structured table that defines strict column widths and text justification.

To define a line in the formatted ASCII output, include the **line** statement within the **format** statement. Within the **line** statement, include **field** statements that map the leaf nodes in the output tree to fields in the line. The **field** statement's argument is the leaf identifier. Fields must be emitted in the same order as you defined the leaf statements.

The CLI output for the following RPC is a single line with three values. Note that you can include other ODL statements within the **field** and **line** statements to customize the formatting for either a single field or all fields within that line, respectively.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf my-version {
            type string;
            description "Version";
         }
         leaf my-model {
            type string;
            description "Model";
         }
         leaf comment {
            type string;
            description "Comment";
         }
         junos-odl:format xyz-information-format {
            junos-odl:comma;
            junos-odl:space;
            junos-odl:line {
               junos-odl:field "my-version" {
                  junos-odl:capitalize;
               }
               junos-odl:field "my-model";
               junos-odl:field "comment";
            }
         }
      }
   }
}
```

fieldwrap and wordwrap

The **fieldwrap** and **wordwrap** statements enable you to more logically wrap content when a line's width is greater than the width of the display. By default, content that extends past the edge of the display wraps at the point where it meets the right margin, without concern for word boundaries.

The **fieldwrap** statement wraps a field's complete contents to the next line when the current line is so long that it extends past the right edge of the display. If you do not use this statement, the string wraps automatically but without regard for appropriate word breaks or the prevailing margin.

Consider the following lines of output:

```
Output errors:
Carrier transitions: 1, Errors: 0, Collisions: 0, Drops: 0, Aged packets: 0
```

If the display is narrower than usual, the line could wrap in the middle of a word as shown in the following sample output:

```
Output errors:
Carrier transitions: 1, Errors: 0, Collisions: 0, Dro
ps: 0, Aged packets: 0
```

When the **fieldwrap** statement is included for a field, the entire field is moved to the next line.

```
Output errors:
Carrier transitions: 1, Errors: 0, Collisions: 0,
Drops: 0, Aged packets: 0
```

The **wordwrap** statement is only used on the rightmost column in a table to wrap sections of a multiword value to subsequent lines when the current line is too long. This effectively creates a column of text. In the following example, the **wordwrap** statement divides the description string at word boundaries:

```
Packet type     Total          Last 5 seconds    Description
            Sent   Received    Sent   Received
 Hello         0          0       4          5   Establish and maintain
                                                  neighbor relationships.
   DbD        20         25       0          0   (Database description packets)
                                                  Describe the contents of
                                                  the topological database.
 LSReq         6          5       0          0   (Link-State Request packets)
                                                  Request a precise instance
                                                  of the database.
```

To improve the wrapping behavior in the RPC's formatted ASCII output, include the **fieldwrap** statement in each field's formatting instructions. To wrap the rightmost column in a table, include the **wordwrap** statement in the rightmost field's formatting instructions.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            type string;
            description "Version";
         }
         leaf desc {
            type string;
            description "Description";
         }
         junos-odl:format xyz-information-format {
            junos-odl:picture "@<<<<<<<<<<<<<<@<<<<<<<<<<<<<<";
            junos-odl:line {
               junos-odl:field "version" {
                   junos-odl:fieldwrap;
               }
               junos-odl:field "desc" {
                   junos-odl:wordwrap;
               }
            }
         }
      }
   }
}
```

float, header, picture, and truncate

You can create tables in the RPC's formatted ASCII output by defining a **header** statement, a **picture** statement, and one or more **line** statements. The optional **header** statement defines the column headings for a table, but it can also just define general text. The **picture** statement graphically depicts the placement, justification, and width of the columns in a table. The **line** and **field** statements define the table rows and their fields.

The argument for the **picture** statement is a string that includes the at (@), less than (<), greater than (>), and vertical bar (|) symbols to define the placement, justification, and width of the table columns. The @ symbol defines the leftmost position in a column that a value in a field can occupy. The <, >, and | symbols indicate left, right, and center justification, respectively. Repeating the <, >, or | symbol defines the column width. Table 14 on page 356 summarizes the symbols. You can also insert one or more blank spaces between columns.

Table 14: picture Statement Symbols

Symbol	Description
@	Defines the leftmost position in a column that a value in a field can occupy.
|	Centers the contents of the field. Repeated symbols define the column width.

Table 14: picture Statement Symbols (continued)

Symbol	Description
<	Left justifies the contents of the field. Repeated symbols define the column width.
>	Right justifies the contents of the field. Repeated symbols define the column width.

The following **picture** statement defines a left-justified column, a centered column, and a right-justified column that are each six characters wide and separated by a single space:

```
junos-odl:picture "    @<<<<< @||||| @>>>>>";
```

To define a table row, include the **line** statement, and map leaf nodes to fields in the line. The **field** statement's argument is the leaf identifier.

```
junos-odl:line {
   junos-odl:field "slot";
   junos-odl:field "state";
   junos-odl:field "comment";
}
```

When a table field must include one of several mutually exclusive values, you can repeat the @ symbol in the **picture** statement for each potential value and include the **float** statement within the **field** statement for each mutually exclusive value after the first value. Then if the first element does not have a value, subsequent possible elements with the **float** statement are tested until a value is returned. The value floats into the position defined by the first @ symbol instead of leaving a blank field.

For example, the following **picture** statement causes the output to include one of two mutually exclusive values in the second column:

```
junos-odl:picture "    @<<<<<  @@<<<<<";
junos-odl:line {
   junos-odl:field "slot";
   junos-odl:field "state";
   junos-odl:field "comment"{
      junos-odl:float;
   }
}
```

You can also use the **float** statement when you know a tag corresponding to a specific table field might be missing in certain situations, and you want to eliminate the extra blank space.

The **truncate** statement guarantees that a field's value does not exceed the width of the column defined by the **picture** statement. The **truncate** statement causes the output to omit any characters in the node's value that would cause it to exceed the width of the field. If the **truncate** statement is omitted, and the output exceeds the width of the field, the complete contents are displayed, which might distort the table. You should use this

statement with care, particularly with numbers, because the output does not provide any indication that the value is truncated.

The CLI formatting for the following RPC defines a small table with two columns. The **comment** field includes the **float** and **truncate** statements. If the **state** output element contains a value, the value is placed in the second column. However, if the **state** output element is empty, the value for the **comment** node, if one exists, is included in the table and moved into the second column. If the comment exceeds the width of that column, it is truncated to fit the column width.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf slot {
            type string;
            description "Slot number";
         }
         leaf state {
            type string;
            description "State";
         }
         leaf comment {
            type string;
         }
         junos-odl:format xyz-information-format {
            junos-odl:header "Slot   State       \n";
            junos-odl:picture "@<<<<< @@||||||||||||||||||";
            junos-odl:line {
               junos-odl:field "slot";
               junos-odl:field "state";
               junos-odl:field "comment"{
                  junos-odl:float;
                  junos-odl:truncate;
               }
            }
         }
      }
   }
}
```

format

When you execute an RPC on a device running Junos OS, it returns the RPC reply as an XML document. Container and leaf nodes under the RPC **output** statement translate into XML elements in the RPC reply. In YANG RPCs for devices running Junos OS, you can also define custom formatted ASCII output that is displayed when you execute the RPC on the Junos OS command-line interface (CLI) or request RPC output in text format.

Starting in Junos OS Release 17.3, to create custom command output for a specific RPC output container, define the **format** statement. The **format** statement defines the CLI formatting for the parent container and is not included as a node in the XML data of the RPC reply. Within the **format** statement, map the data for the parent container to output fields, and use statements from the Junos OS ODL extensions module to specify how to display the output for that parent container.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         // leaf definitions
         junos-odl:format xyz-information-format {
            // CLI formatting for the parent container
         }
      }
   }
}
```

To create custom command output for a specific RPC output container in Junos OS Release 17.2 and earlier releases, see "cli-format" on page 351.

header and header-group

The **header** statement enables you to define a header string that precedes a set of fields in the RPC's formatted ASCII output, and the **header-group** statement causes only the first header string to be emitted when two or more headers in the same header group would be included in the output.

To define a header string and associate it with a header group, include the **header** and **header-group** statements, respectively, within the **format** statement. The **header-group** argument is a user-defined string that identifies a particular header group. Every **format** statement that includes the **header-group** statement with the same identifier belongs to the same header group. The following example defines a **format** statement associated with the header group **color-tags**.

```
junos-odl:format red-format {
   junos-odl:header-group "color-tags";
   junos-odl:header "Color tags\n";
   ...
}
```

When multiple **format** statements are associated with the same header group, and the tags emitted by two or more of those statements are present in the output, the CLI output only emits the first header it encounters and suppresses any subsequent headers belonging to that header group.

To emit only the first header string for a header group in the RPC's CLI output, include the **header-group** statement and identifier in all **format** statements belonging to that header group. The following sample RPC **output** statement associates two containers and their format statements with the header group **color-tags**.

```
output {
   container red-group {
      container red {
         leaf redtag1 {
            type string;
         }
         leaf redtag2 {
            type string;
         }
         junos-odl:format red-format {
            junos-odl:header-group "color-tags";
            junos-odl:header "Color tags\n";
```

```
            junos-odl:picture "@<<<<<<<<<<<  @<<<<<<<<<<<";
            junos-odl:indent 5;
            junos-odl:line {
               junos-odl:field "redtag1";
               junos-odl:field "redtag2";
            }
         }
      }
   }
   container blue-group {
      container blue {
         leaf bluetag1 {
            type string;
         }
         leaf bluetag2 {
            type string;
         }
         junos-odl:format blue-format {
            junos-odl:header-group "color-tags";
            junos-odl:header "Color tags\n";
            junos-odl:picture "@<<<<<<<<<<<  @<<<<<<<<<<<";
            junos-odl:indent 5;
            junos-odl:line {
               junos-odl:field "bluetag1";
               junos-odl:field "bluetag2";
            }
         }
      }
   }
}
```

Consider an RPC reply with the following XML tags:

```
<rpc-reply>
    <red-group>
        <red>
          <redtag1>red-1</redtag1>
          <redtag2>red-2</redtag2>
        </red>
    </red-group>
    <blue-group>
        <blue>
          <bluetag1>blue-1</bluetag1>
          <bluetag2>blue-2</bluetag2>
        </blue>
    </blue-group>
</rpc-reply>
```

When the RPC reply is rendered in the CLI and the same **header-group** statement is present in each **format** statement, only the first header string is emitted in the output, which in this case is the header string defined in the **format** statement with the identifier **red-format**.

```
Color tags
      red-1          red-2
      blue-1         blue-2
```

If you omit the **header-group** statement from the **format** statement, the header string defined for each set of fields is included in the output.

```
Color tags
      red-1          red-2
Color tags
      blue-1         blue-2
```

indent

The **indent** statement causes all of the lines in the scope of the statement other than the header row to be indented by the specified number of characters.

To indent lines, include the **indent** statement and the number of spaces to indent the lines at the top level of the **format** statement The formatted ASCII output for the following RPC displays a line that is indented by 10 spaces in the output.

```
   rpc get-xyz-information {
      output {
         container xyz-information {
            leaf version {
               type string;
               description "Version";
            }
            leaf model {
               type string;
               description "Model";
            }
            junos-odl:format xyz-information-format {
               junos-odl:header "xyz information\n";
               junos-odl:indent 10;
               junos-odl:line {
                  junos-odl:field "version";
                  junos-odl:field "model";
               }
            }
         }
      }
   }
```

When you execute the RPC, the header is left justified, and the line containing the two fields is indented ten spaces.

```
xyz information
          version model
```

no-line-break

The **no-line-break** statement is used to display multiple values on the same line in the case where the output emits multiple entities with the same tag names. When you include

the **no-line-break** statement, repeated formats are placed on the same line. If you omit the statement, repeated formats are placed on separate lines.

For example, you might want to display all SONET errors together on the same line.

```
SONET errors:
BPI-B1 0 BIP-B2 0 REI-L 0 BIP-B3 0 REI-P 0
```

To place the tags for multiple entities within the same line of output, include the **no-line-break** statement in the **format** statement for that container.

```
rpc get-sonet-errors {
   output {
      container sonet-error-information {
         container sonet-errors {
            leaf sonet-error-name {
               type string;
               description "SONET error name";
            }
            leaf sonet-error-count {
               type integer;
               description "SONET error count";
            }
            junos-odl:format sonet-errors-format {
               junos-odl:no-line-break;
               junos-odl:space;
               junos-odl:header "SONET errors:\n";
               junos-odl:line {
                  junos-odl:field "sonet-error-name";
                  junos-odl:field "sonet-error-count";
               }
            }
         }
      }
   }
}
```

If the RPC output returns multiple entities, the output places each repeated set of fields on the same line.

```
SONET errors:
BPI-B1 0 BIP-B2 0 REI-L 0 BIP-B3 0 REI-P 0
```

If you omit the **no-line-break** statement, the output places each repeated set of fields on its own line.

```
SONET errors:
BPI-B1 0
BIP-B2 0
REI-L 0
BIP-B3 0
REI-P 0
```

space

The **space** statement appends a space to the node's value in the RPC's formatted ASCII output. For example:

```
value1 value2 Label3: value3
```

The **space** statement is often used in conjunction with the **comma** statement to delimit fields in a line of output with a comma followed by a space.

To generate a space after a value in the output field, include the **space** statement within the **format** statement. The placement of a statement determines the statement's scope. Placing the statement within a single field generates a space after that field only.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            type string;
            description "Version";
         }
         // additional leaf definitions
         junos-odl:format xyz-information-format {
            junos-odl:space;
            junos-odl:line {
               junos-odl:field "version";
               // additional fields
            }
         }
      }
   }
}
```

style

The **style** statement defines one of several formats for the RPC output. For detailed information about using the **style** statement to create different levels of output, see "Defining Different Levels of Output in Custom YANG RPCs for Devices Running Junos OS" on page 365.

template

The **template** statement explicitly defines the format for an output field for a given node, including the output string and placement of the node's value within the string. If the **template** statement is defined for a leaf node, the corresponding output field automatically uses the template string.

To create a template string for a node, you must include the **template** statement in the definition of the node, and define the string. The placeholders **%s** and **%d** within the string define the type and placement of the node's value. Use **%s** to insert a string value, and **%d** to insert an integer value. For example:

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            junos-odl:template " Version: %s";
            type string;
            description "Version";
         }
      }
   }
}
```

If you define a **template** statement for a node, the output field for that node automatically uses the template text.

```
rpc get-xyz-information {
   output {
      container xyz-information {
         leaf version {
            junos-odl:template " Version: %s";
            type string;
            description "Version";
         }
         junos-odl:format xyz-information-format {
            junos-odl:line {
               junos-odl:field "version";
               // additional fields
            }
         }
      }
   }
}
```

When you execute the RPC, the template is used in the output for that field.

```
Version: value
```

NOTE: If a leaf statement defines both a template and a formal-name statement, and the leading statement is included in the formatting instructions for that field, the output uses the text defined for the formal-name statement and not the text defined for the template statement.

Release History Table

Release	Description
17.3R1	Starting in Junos OS Release 17.3, the CLI formatting for a custom RPC is defined within the **junos-odl:format** extension statement. In earlier releases, the CLI formatting is defined using a container that includes the **junos-odl:cli-format** statement.

Related Documentation

- Understanding the YANG Modules for Junos OS Operational Commands on page 288
- Creating Custom RPCs in YANG for Devices Running Junos OS on page 317
- Defining Different Levels of Output in Custom YANG RPCs for Devices Running Junos OS on page 365
- Understanding Junos OS YANG Extensions for Formatting RPC Output on page 345

Defining Different Levels of Output in Custom YANG RPCs for Devices Running Junos OS

Defining Different Levels of Output in Custom YANG RPCs

You can define custom RPCs for devices running Junos OS using YANG. The RPC output can be customized to emit different data and CLI formatting depending on the RPC input. This enables you to create different styles, or levels of output, for the same RPC.

You can request the desired style by including the appropriate value for the input argument when you invoke the RPC. The action script must process this argument and emit the XML output for the requested style. Junos OS then translates the XML into the corresponding CLI output defined for that style. The RPC template presented in this topic creates two styles: **brief** and **detail**.

To create different styles for the output of an RPC:

1. In the YANG module that includes the RPC, import the Junos OS ODL extensions module, which defines YANG extensions that you use to precisely specify how to render the output when you execute the RPC's command in the CLI or when you request the RPC output in text format.

```
import junos-extension-odl {
   prefix junos-odl;
}
```

NOTE: Starting in Junos OS Release 17.4R1, the Junos OS YANG modules use a new naming convention for the module's name, filename, and namespace.

2. In the RPC input parameters, include a **leaf** statement with type **enumeration**, and use an **enum** statement to define a name for each style.

```
rpc rpc-name {
   description "RPC description";
   junos:command "cli-command" {
      junos:action-execute {
         junos:script "action-script-filename";
      }
   }

   input {
      leaf level {
         type enumeration {
            enum brief {
               description "Display brief output";
            }
            enum detail {
               description "Display detailed output";
            }
         }
      }
   }
}
```

NOTE: Starting in Junos OS Release 17.3, the action-execute statement is a substatement to command. In earlier releases, the action-execute and command statements are placed at the same level, and the command statement is optional.

3. In the RPC **output** statement, create separate **junos-odl:style** statements that define the CLI formatting for each style. The identifier for each **style** statement should match one of the style names defined within the enumerated **leaf** statement.

```
    output {
       container output-container {

          //  leaf definitions

          junos-odl:style brief {
             junos-odl:format output-container-format-brief {
                // formatting for brief output
             }
          }
          junos-odl:style detail {
             junos-odl:format output-container-format-detail {
                // formatting for detailed output
             }
          }

       }
    }
```

NOTE: Starting in Junos OS Release 17.3, the CLI formatting for a custom RPC is defined within the junos-odl:format extension statement, and junos-odl:format is a substatement to junos-odl:style. In earlier releases, the CLI formatting is defined using a container that includes the junos-odl:cli-format statement, and the junos-odl:style statement is included within that container.

4. In the RPC's action script, process the input argument, and emit the XML output for the requested style enclosed in a parent element with a tag name identical to the style name.

```
#!/usr/bin/python
import sys

args = {'level': 'brief'}

# Retrieve user input and store the values in the args dictionary
for arg in args.keys():
    if arg in sys.argv:
        index = sys.argv.index(arg)
        args[arg] = sys.argv[index+1]

print ("<output-container>")
print ("<{}>".format(args['level']))     # tag name is brief or detail

if args['level'] == "brief":
   // print statements for brief output

if args['level'] == "detail":
  // print statements for detailed output
```

```
print ("</{}>".format(args['level']))
print ("</output-container>")
```

The following code outlines the general structure of the RPC and enclosing module. When you invoke the RPC in the CLI and include the input argument **level** and specify either **brief** or **detail**, Junos OS renders the output defined for that style.

```
module module-name {
   namespace "http://yang.juniper.net/yang/1.1/jrpc";
   prefix jrpc;

   import junos-extension {
      prefix junos;
   }
   import junos-extension-odl {
      prefix junos-odl;
   }

   organization
      "Juniper Networks, Inc.";
   description
      "Junos OS YANG module for custom RPCs";

   rpc rpc-name {
      description "RPC description";

      junos:command "cli-command" {
         junos:action-execute {
            junos:script "action-script-filename";
         }
      }

      input {
         leaf level {
            type enumeration {
               enum brief {
                  description "Display brief output";
               }
               enum detail {
                  description "Display detailed output";
               }
            }
         }
      }
      output {
         container output-container {

            // leaf definitions

            junos-odl:style brief {
               junos-odl:format output-container-format-brief {
                  // formatting for brief output
               }
            }

            junos-odl:style detail {
               junos-odl:format output-container-format-detail {
```

```
                // formatting for detailed output
            }
        }

    }
  }
 }
}
```

To execute the RPC in the CLI, issue the command defined by the **junos:command** statement, and specify the style by including the appropriate command-line argument, which in this example is **level**.

```
user@host> cli-command level brief
```

Example: Defining Different Levels of Output

This example presents a simple custom YANG RPC and action script that determine if a host is reachable and print different levels of output depending on the user input.

Requirements

This example uses the following hardware and software components:

- Device running Junos OS Release 17.3R1 or later that supports custom YANG modules.

Overview of the RPC and Action Script

The YANG module presented in this section defines a custom RPC to ping the specified host and return the result using different levels of output based on the user's input. The YANG module **rpc-style-test** is saved in the **rpc-style-test.yang** file. The module imports the Junos OS extension modules, which provide the extensions required to execute custom RPCs on the device and to customize the CLI output.

The module defines the **get-host-status** RPC. The **<get-host-status>** request tag is used to remotely execute the RPC on the device. In the RPC definition, the **junos:command** statement defines the command that is used to execute the RPC in the CLI, which in this case is **show host-status**.

```
    rpc get-host-status {
       description "RPC example to retrieve host status";

       junos:command "show host-status" {
          junos:action-execute {
             junos:script "rpc-style-test.py";
          }
```

```
    }
...
```

The **junos:action-execute** and **junos:script** statements define the action script that is invoked when the RPC is executed. This example uses a Python action script named **rpc-style-test.py** to retrieve the information required by the RPC and return the XML output elements for each level of output as defined in the RPC **output** statement.

NOTE: Starting in Junos OS Release 17.3, the action-execute statement is a substatement to command. In earlier releases, the action-execute and command statements are placed at the same level, and the command statement is optional.

The RPC has two input parameters, **hostip** and **level**. The **hostip** parameter is the host to check for reachability. The **level** parameter is used to select between the different styles for the RPC output. When the user executes the RPC, the user includes the target host's IP address and a level, **brief** or **detail**. The action script defines the default value for **level** as **'brief'**, so if the user omits this argument, the RPC will print the output corresponding to the brief style.

```
input {
   leaf hostip {
      description "Host IP address";
      type string;
   }
   leaf level {
      type enumeration {
         enum brief {
            description "Display brief output";
         }
         enum detail {
            description "Display detailed output";
         }
      }
   }
}
```

The RPC also defines the output nodes that must be emitted by the corresponding action script. The root node is the **<host-status-information>** element, which encloses either the **<brief>** or the **<detail>** element, depending on the user input, along with the child output nodes specified for each level of output. Both levels of output include the **<hostip>** and **<status>** child elements, but the **<detail>** element also includes the **<date>** child element. The **junos-odl:format** statements define the formatting for the output that is displayed in the CLI. This node is not emitted in the output XML tree.

```
output {
   container host-status-information {
      ...
      junos-odl:style brief {
         junos-odl:format host-status-information-format-brief {
            ...
         }
```

```
            }
            junos-odl:style detail {
               junos-odl:format host-status-information-format-detail {
                  ...
               }
            }
         }
      }
   }
}
```

The action script pings the host to determine if it is reachable and sets the status based on the results. The script then constructs and prints the XML for the RPC output based on the specified **level** argument. The XML tree must exactly match the hierarchy defined in the RPC.

The module containing the RPC and the action script file are added to the device as part of a new YANG package named **rpc-style-test**.

YANG Module

The YANG module, **rpc-style-test.yang**, defines the RPC, the command used to execute the RPC in the CLI, and the name of the action script to invoke when the RPC is executed. The base name of the file must match the module name.

```
/*
* Copyright (c) 2014 Juniper Networks, Inc.
* All rights reserved.
*/

module rpc-style-test {
  namespace "http://yang.juniper.net/yang/1.1/jrpc";
  prefix jrpc;

  import junos-extension-odl {
    prefix junos-odl;
  }
  import junos-extension {
    prefix junos;
  }

  organization
    "Juniper Networks, Inc.";

  description
    "Junos OS YANG module for RPC example";

  rpc get-host-status {
     description "RPC example to retrieve host status";

     junos:command "show host-status" {
        junos:action-execute {
           junos:script "rpc-style-test.py";
        }
     }

     input {
        leaf hostip {
           description "Host IP address";
           type string;
```

```
         }
         leaf level {
            type enumeration {
               enum brief {
                  description "Display brief output";
               }
               enum detail {
                  description "Display detailed output";
               }
            }
         }
      }
      output {
         container host-status-information {
            leaf hostip {
               type string;
               description "Host IP";
            }
            leaf status {
               type string;
               description "Operational status";
            }
            leaf date {
               type string;
               description "Date information";
            }
            junos-odl:style brief {
               junos-odl:format host-status-information-format-brief {
                  junos-odl:header "Brief output\n";
                  junos-odl:picture "@<<<<<<<<<<<< @";
                  junos-odl:space;
                  junos-odl:line {
                    junos-odl:field "hostip";
                    junos-odl:field "status";
                  }
               }
            }
            junos-odl:style detail {
               junos-odl:format host-status-information-format-detail {
                  junos-odl:header "Detail output\n";
                  junos-odl:picture "@<<<<<<<<<<<<  @<<<<<<<<<<<< @";
                  junos-odl:space;
                  junos-odl:line {
                     junos-odl:field "hostip";
                     junos-odl:field "status";
                     junos-odl:field "date";
                  }
               }
            }
         }
      }
   }
}
```

The corresponding action script is **rpc-style-test.py**.

Action Script The following action script prints different levels of output based on the value of the **level** argument. The script defines a default value of **'brief'** for the **level** argument so that if the user omits the argument, the brief style of output is the default.

```
#!/usr/bin/python
import re, sys
import os

args = {'hostip': None, 'level': 'brief'}

# Retrieve user input and store the values in the args dictionary
for arg in args.keys():
    if arg in sys.argv:
        index = sys.argv.index(arg)
        args[arg] = sys.argv[index+1]

f = os.popen('date')
now = f.read()

# Ping target host and set the status
if args['hostip'] is not None:
    response = os.system('ping -c 1 ' + args['hostip'] + ' > /dev/null')
    if response == 0:
        pingstatus = "Host is Active"
    else:
        pingstatus = "Host is Inactive"
else:
    pingstatus = "Invalid host"

# Print RPC XML for the given style
print ("<host-status-information>")
print ("<{}>".format(args['level']))
print ("<hostip>{}</hostip>".format(args['hostip']))
print ("<status>{}</status>".format(pingstatus))
if args['level'] == "detail":
    print ("<date>{}</date>".format(now))
print ("</{}>".format(args['level']))
print ("</host-status-information>")
```

Loading the RPC on the Device

To add the RPC and action script to the Junos OS schema on a device running Junos OS:

1. Download the YANG module and action script to the device running Junos OS.

2. Ensure that the Python action script meets the following requirements:

   - File owner is either root or a user in the Junos OS **super-user** login class.
   - Only the file owner has write permission for the file.

3. (Optional) Validate the syntax for the YANG module and action script.

   user@host> **request system yang validate module /var/tmp/rpc-style-test.yang action-script /var/tmp/rpc-style-test.py**

   ```
   YANG modules validation : START
   YANG modules validation : SUCCESS
   ```

```
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
```

4. Add the YANG module and action script to a new YANG package.

```
user@host> request system yang add package rpc-style-test module
/var/tmp/rpc-style-test.yang action-script /var/tmp/rpc-style-test.py
YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
TLV generation: START
TLV generation: SUCCESS
Building schema and reloading /config/juniper.conf.gz ...
Activating /config/juniper.conf.gz ...
mgd: commit complete
Restarting mgd ...
```

5. When the system prompts you to restart the Junos OS CLI, press **Enter** to accept the default value of **yes**, or type **yes** and press **Enter**.

```
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes) yes

Restarting cli ...
```

Enabling Execution of Python Scripts

To enable the device to execute unsigned Python scripts, configure the **language python** statement.

```
[edit]
user@host# set system scripts language python
user@host# commit and-quit
```

Verifying the RPC

Purpose Verify that the RPC is working as expected.

Action From operational mode, execute the RPC in the CLI by issuing the command defined by the **junos:command** statement in the RPC definition, and include the **hostip** input argument, and include the **level** argument for each different level of output.

```
user@host> show host-status hostip 198.51.100.1 level brief
```

```
Brief output
198.51.100.1   Host is Active
```

You can view the corresponding XML by appending **| display xml** to the command.

```
user@host> show host-status hostip 198.51.100.1 level brief | display xml
```

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.3R1/junos">
    <host-status-information>
        <brief>
            <hostip>
                198.51.100.1
            </hostip>
            <status>
                Host is Active
            </status>
        </brief>
    </host-status-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Similarly, for the detailed output:

```
user@host> show host-status hostip 198.51.100.10 level detail
```

```
Detail output
198.51.100.10   Host is Inactive Fri Feb  8 11:55:54 PST 2019
```

```
user@host> show host-status hostip 198.51.100.10 level detail | display xml
```

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.3R1/junos">
    <host-status-information>
        <detail>
            <hostip>
                198.51.100.10
            </hostip>
            <status>
                Host is Inactive
            </status>
            <date>
                Fri Feb  8 16:03:35 PST 2019
            </date>
        </detail>
    </host-status-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```


Meaning When the RPC is executed, the action script is invoked. The action script prints the XML hierarchy for the given level of output as defined in the RPC **output** statement. When the RPC is executed in the CLI, the device uses the CLI formatting defined in the RPC to convert the XML output into the displayed CLI output.

Related Documentation

- Understanding the YANG Modules for Junos OS Operational Commands on page 288
- Creating Custom RPCs in YANG for Devices Running Junos OS on page 317
- Understanding Junos OS YANG Extensions for Formatting RPC Output on page 345
- Customizing YANG RPC Output on Devices Running Junos OS on page 348

Displaying Valid Command Option and Configuration Statement Values in the CLI for Custom YANG Modules

Certain devices running Junos OS enable you to load custom YANG modules on the device to add data models that are not natively supported by Junos OS. When you add custom YANG data models to a device, you must also supply an action or translation script that handles the translation logic between the YANG data model and Junos OS. Although the script logic can ensure that a user supplies valid values for a given command option or configuration statement, that logic is not always transparent to the user. Starting in Junos OS Release 19.2R1, the CLI can display the set of possible values for certain command options or configuration statements in a custom YANG data model by including the **action-expand** extension statement in the option or statement definition and referencing a script that handles the logic.

- Understanding Context-Sensitive Help for Custom YANG Modules on page 376
- Defining the YANG Module on page 377
- Creating the CLI Expansion Script on page 378
- Loading the YANG Package on page 380
- Example: Displaying Context-Sensitive Help for a Command Option on page 381

Understanding Context-Sensitive Help for Custom YANG Modules

In operational or configuration mode, you can get context-sensitive help when you type a question mark (?) anywhere in the Junos OS command-line interface (CLI). When you execute a command or configure a device, the CLI's context-sensitive help displays the valid options and option values for a command or the valid configuration statements and leaf statement values in the configuration statement hierarchy. Additionally, context-sensitive help shows the possible completions for incomplete option names, statement names, and their values.

The CLI can also display the values that are valid for certain command options or configuration statements in a custom YANG data model. The CLI can display all possible values or a subset of values that match on partial input from the user. For example:

```
user@host> show host-status hostip ?
Possible completions:
  <hostip>              Host IP address
  10.10.10.1            IPv4 address
  10.10.10.2            IPv4 address
  172.16.0.1            IPv4 address
  198.51.100.1          IPv4 address
  198.51.100.10         IPv4 address
  2001:db8::1           IPv6 address (DC 1...128)
  2001:db8::fdd2        IPv6 address (DC 1...128)
```

```
user@host> show host-status hostip 198?
Possible completions:
  <hostip>              Host IP address
  198.51.100.1          IPv4 address
  198.51.100.10         IPv4 address
```

To display the set of valid values for a given command option or configuration statement in a custom YANG module:

1. Define the **action-expand** and **script** extension statements under the appropriate input parameter or configuration statement in the YANG module as described in "Defining the YANG Module" on page 377.
2. Create a Python script that checks for user input, calculates the possible values of the command option or configuration statement, and sends the appropriate output to the CLI, as described in "Creating the CLI Expansion Script" on page 378.

   NOTE: **The CLI expansion script only displays the valid values in the CLI. The module's translation script or action script must still include the logic that ensures that only valid values are accepted and processed.**

3. Load the YANG module, any translation or action scripts, and the CLI expansion script as part of a custom YANG package on the device as described in "Loading the YANG Package" on page 380.

Defining the YANG Module

To indicate that the CLI should display the set of valid values for a given command option or configuration statement when the user requests context-sensitive help, your YANG module must:

- Import the Junos OS DDL extensions module.
- Include the **action-expand** extension statement and **script** substatement in the corresponding command option or configuration statement definition.

You can include the **action-expand** statement within a **leaf** statement in modules that define custom RPCs and within a **leaf** or **leaf-list** statement in modules that define custom

configuration hierarchies. The **script** statement should reference the Python script that defines your custom logic.

For example, in the following module, the RPC defines the **hostip** input parameter, which calls the **hostip-expand.py** Python script when the user requests context-sensitive help for the **hostip** argument in the CLI. The script implements the custom logic that displays the valid values for that argument in the CLI.

```
module rpc-host-status {
  namespace "http://yang.juniper.net/examples/rpc-cli";
  prefix jrpc;

  import junos-extension-odl {
    prefix junos-odl;
  }
  import junos-extension {
    prefix junos;
  }

  rpc get-host-status {
     description "RPC example to retrieve host status";

     junos:command "show host-status" {
        junos:action-execute {
           junos:script "rpc-host-status.py";
        }
     }

     input {
        leaf hostip {
           description "Host IP address";
           type string;
           junos:action-expand {
              junos:script "hostip-expand.py";
           }
        }
        leaf level {
           type enumeration {
              enum brief {
                 description "Display brief output";
              }
              enum detail {
                 description "Display detailed output";
              }
           }
        }
     }
     output {
     ...
     }
  }
}
```

Creating the CLI Expansion Script

When you define the **action-expand** statement and **script** substatement for a command option or configuration statement in a custom YANG module and you request context-sensitive help for that option or statement value in the CLI, the referenced Python

script is invoked. The script must contain the custom logic that calculates and displays all possible values for that parameter or displays a subset of values that match on partial input from the user.

For example, the following command should display all valid values for the **hostip** argument:

```
user@host> show host-status hostip ?
```

And the following command should display all valid values that start with "198":

```
user@host> show host-status hostip 198?
```

To display the valid values for a command option or configuration statement in the CLI, the Python script should perform the following functions:

- Import the **jcs** library along with any other required Python libraries.
- Retrieve and process any user input.

  If you specify partial input for an option or statement value in the CLI, the script's command-line arguments include the **symbol** argument, which is a string containing the user input.
- Define or calculate the valid values for the parameter.
- Call the **jcs.expand()** function for each value to display on the command line.

The script calls the **jcs:expand()** function for each option or statement value that gets displayed in the CLI. The syntax for the **jcs:expand()** function is:

```
jcs.expand(value, description, <units>, <range>)
```

Where:

value—String defining a valid value for the given command option or configuration statement.

description—String that describes the value.

units—(Optional) String that defines the units for the corresponding value.

range—(Optional) String that defines the range for the corresponding value.

For each call to the **jcs.expand()** function, the script emits the value, description, units, and range that are provided in the function arguments in the CLI. For example, given the following call to **jcs.expand()** in the script:

```
jcs.expand("2001:db8:4136::fdd2", "IPv6 address", "DC", "1...128")
```

The corresponding CLI output is:

```
Possible completions:
  <hostip>              Host IP address
  2001:db8:4136::fdd2   IPv6 address (DC 1...128)
```

The following sample script first checks for the presence of **symbol** in the script's command-line arguments, and if present, sets the **symbol** variable equal to the user's input. The script then calculates the set of valid values for the parameter based on the user's input, and calls the **jcs.expand()** function for each value to display in the CLI.

```
#!/usr/bin/python
import sys
import os
import jcs

symbol = ""

# Retrieve user input in symbol argument and store the value
if "symbol" in sys.argv:
    index = sys.argv.index("symbol")
    symbol = sys.argv[index+1]

description_ipv4 = "IPv4 address"
description_ipv6 = "IPv6 address"
expand_colon = ":"
expand_units = "DC"
expand_range = "1...128"

item = ["10.10.10.1", "10.10.10.2", "2001:db8::1",
        "172.16.0.1", "198.51.100.1", "198.51.100.10", "2001:db8::fdd2"]

for i in range(len(item)):
    if item[i].startswith(symbol) or not symbol:
        if not expand_colon in item[i]:
            jcs.expand(item[i], description_ipv4)
        else:
            jcs.expand(item[i], description_ipv6,
            expand_units, expand_range)
```

The CLI expansion script only displays the valid values, units, and ranges for the command option or configuration statement in the CLI. The module's translation script or action script must ensure that only valid values are accepted and processed.

Loading the YANG Package

When you load a YANG package on a device running Junos OS, include any CLI expansion scripts in the list of action scripts for that package. Junos OS automatically copies the script to the **/var/db/scripts/action** directory.

To load a new package and include custom CLI expansion scripts:

1. Ensure that the Python scripts meet the following requirements:
   - File owner is either root or a user in the Junos OS **super-user** login class.
   - Only the file owner has write permission for the file.
2. In configuration mode, enable the device to execute unsigned Python scripts.

```
[edit]
user@host# set system scripts language python
user@host# commit and-quit
```

3. In operational mode, load the YANG package and include the CLI expansion script in the **action-script** list.

```
user@host> request system yang add package rpc-host-status module
/var/tmp/rpc-host-status.yang action-script [/var/tmp/rpc-host-status.py
/var/tmp/hostip-expand.py]
YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
TLV generation: START
TLV generation: SUCCESS
Building schema and reloading /config/juniper.conf.gz ...
Activating /config/juniper.conf.gz ...
mgd: commit complete
Restarting mgd ...
```

4. When the system prompts you to restart the Junos OS CLI, press **Enter** to accept the default value of **yes**.

```
...
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes)

Restarting cli ...
```

NOTE: To prevent CLI-related or configuration database errors, we recommend that you do not perform any CLI operations or change the configuration while a device is in the process of adding, updating, or deleting a YANG package and modifying the schema.

Example: Displaying Context-Sensitive Help for a Command Option

This example presents a custom YANG module that uses the **action-expand** extension statement and a custom script to display the set of possible values for one of the command options when a user requests context-sensitive help in the CLI for that option.

Requirements

This example uses the following hardware and software components:

- Device running Junos OS Release 19.2R1 or later that supports loading custom YANG data models.

Overview of the YANG Module and Scripts

The YANG module presented in this section defines a custom RPC to ping the specified host and return the result. The YANG module **rpc-host-status** is saved in the **rpc-host-status.yang** file. The module imports the Junos OS extension modules, which provide the extensions required to execute custom RPCs on the device and to customize the output and context-sensitive help in the CLI.

The module defines the **get-host-status** RPC. The **junos:command** statement defines the command that is used to execute the RPC in the CLI, which in this case is **show host-status**. The **junos:action-execute** and **junos:script** statements define the action script that is invoked when the RPC is executed.

```
rpc get-host-status {
   description "RPC example to retrieve host status";

   junos:command "show host-status" {
      junos:action-execute {
         junos:script "rpc-host-status.py";
      }
   }
```

The **hostip** input parameter includes the **junos:action-expand** and **junos:script** statements, which define the script that is invoked when the user requests context-sensitive help in the CLI for that input parameter. The **hostip-expand.py** script processes the user's input, which is passed to the script as the argument **symbol**, and displays the set of values that the user can enter for that command option.

```
   input {
      leaf hostip {
         description "Host IP address";
         type string;
         junos:action-expand {
            junos:script "hostip-expand.py";
         }
      }
      ...
   }
```

The expansion script displays the valid values for **hostip** in the CLI. The action script implements the logic that determines if the provided value is valid. The module containing the RPC and the action scripts are added to the device as part of a new YANG package named **rpc-host-status**.

YANG Module

The YANG module, **rpc-host-status.yang**, defines the RPC, the command used to execute the RPC in the CLI, and the name of the action script to invoke when the RPC is executed.

```
/*
* Copyright (c) 2019 Juniper Networks, Inc.
* All rights reserved.
*/

module rpc-host-status {
  namespace "http://yang.juniper.net/examples/rpc-cli";
  prefix jrpc;

  import junos-extension-odl {
    prefix junos-odl;
  }
  import junos-extension {
    prefix junos;
  }

  organization
    "Juniper Networks, Inc.";

  description
    "Junos OS YANG module for RPC example";

  rpc get-host-status {
     description "RPC example to retrieve host status";

     junos:command "show host-status" {
        junos:action-execute {
           junos:script "rpc-host-status.py";
        }
     }

     input {
        leaf hostip {
           description "Host IP address";
           type string;
           junos:action-expand {
              junos:script "hostip-expand.py";
           }
        }
        leaf level {
           type enumeration {
              enum brief {
                 description "Display brief output";
              }
              enum detail {
                 description "Display detailed output";
              }
           }
        }
     }
     output {
        container host-status-information {
           leaf hostip {
              type string;
              description "Host IP";
           }
           leaf status {
```

```
               type string;
               description "Operational status";
            }
            leaf date {
               type string;
               description "Date information";
            }
            junos-odl:style brief {
               junos-odl:format host-status-information-format-brief {
                  junos-odl:header "Brief output\n";
                  junos-odl:picture "@<<<<<<<<<<<< @";
                  junos-odl:space;
                  junos-odl:line {
                    junos-odl:field "hostip";
                    junos-odl:field "status";
                  }
               }
            }
            junos-odl:style detail {
               junos-odl:format host-status-information-format-detail {
                  junos-odl:header "Detail output\n";
                  junos-odl:picture "@<<<<<<<<<<<<  @<<<<<<<<<<<< @";
                  junos-odl:space;
                  junos-odl:line {
                     junos-odl:field "hostip";
                     junos-odl:field "status";
                     junos-odl:field "date";
                  }
               }
            }
         }
      }
   }
}
```

The corresponding action script is **rpc-host-status.py**.

Action Script

```
#!/usr/bin/python
import re, sys
import os

args = {'hostip': None, 'level': 'brief'}
valid_addresses = ["10.10.10.1", "10.10.10.2", "2001:db8::1",
        "172.16.0.1", "198.51.100.1", "198.51.100.10", "2001:db8::fdd2"]

# Retrieve user input and store the values in the args dictionary
for arg in args.keys():
    if arg in sys.argv:
        index = sys.argv.index(arg)
        args[arg] = sys.argv[index+1]

f = os.popen('date')
now = f.read()

# Ping target host and set the status
if args['hostip'] in valid_addresses:
    response = os.system('ping -c 1 ' + args['hostip'] + ' > /dev/null')
    if response == 0:
        pingstatus = "Host is Active"
```

```
    else:
        pingstatus = "Host is Inactive"
else:
    pingstatus = "Invalid host"

# Print RPC XML for the given style
print ("<host-status-information>")
print ("<{}>".format(args['level']))
print ("<hostip>{}</hostip>".format(args['hostip']))
print ("<status>{}</status>".format(pingstatus))
if args['level'] == "detail":
    print ("<date>{}</date>".format(now))
print ("</{}>".format(args['level']))
print ("</host-status-information>")
```

The action script that handles the logic to display the valid values of **hostip** in the CLI is **hostip-expand.py**.

CLI expansion script

```
#!/usr/bin/python
import sys
import os
import jcs

symbol = ""

# Retrieve user input in symbol argument and store the value
if "symbol" in sys.argv:
    index = sys.argv.index("symbol")
    symbol = sys.argv[index+1]

description_ipv4 = "IPv4 address"
description_ipv6 = "IPv6 address"
expand_colon = ":"
expand_units = "DC"
expand_range = "1...128"

item = ["10.10.10.1", "10.10.10.2", "2001:db8::1",
        "172.16.0.1", "198.51.100.1", "198.51.100.10", "2001:db8::fdd2"]

for i in range(len(item)):
    if item[i].startswith(symbol) or not symbol:
        if not expand_colon in item[i]:
            jcs.expand(item[i], description_ipv4)
        else:
            jcs.expand(item[i], description_ipv6,
            expand_units, expand_range)
```

Loading the YANG Module and Scripts on the Device

To add the YANG module and scripts to the device running Junos OS:

1. Download the YANG module and scripts to the device running Junos OS.

2. Ensure that the Python scripts meet the following requirements:

   - File owner is either root or a user in the Junos OS **super-user** login class.

- Only the file owner has write permission for the file.

3. Add the YANG module and scripts to a new YANG package.

```
user@host> request system yang add package rpc-host-status module
/var/tmp/rpc-host-status.yang action-script [ /var/tmp/rpc-host-status.py
/var/tmp/hostip-expand.py ]
YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
TLV generation: START
TLV generation: SUCCESS
Building schema and reloading /config/juniper.conf.gz ...
mgd: commit complete
Restarting mgd ...
```

4. When the system prompts you to restart the Junos OS CLI, press **Enter** to accept the default value of **yes**, or type **yes** and press **Enter**.

```
WARNING: cli has been replaced by an updated version:
...
Restart cli using the new version ? [yes,no] (yes) yes

Restarting cli ...
```

Enabling Execution of Python Scripts

To enable the device to execute unsigned Python scripts, configure the **language python** statement.

```
[edit]
user@host# set system scripts language python
user@host# commit and-quit
```

Verifying the Context-Sensitive Help

Purpose Verify that the CLI expansion script is working as expected.

Action From operational mode, request context-sensitive help in the CLI by issuing the command defined by the **junos:command** statement in the RPC definition, and include the **hostip** input argument and a question mark (?).

```
user@host> show host-status hostip ?
```

```
Possible completions:
  <hostip>             Host IP address
  10.10.10.1           IPv4 address
  10.10.10.2           IPv4 address
  172.16.0.1           IPv4 address
  198.51.100.1         IPv4 address
  198.51.100.10        IPv4 address
  2001:db8::1          IPv6 address (DC 1...128)
  2001:db8::fdd2       IPv6 address (DC 1...128)
```

Perform the same operation with partial user input and verify that the displayed values correctly match the input.

```
user@host> show host-status hostip 198?
```

```
Possible completions:
  <hostip>             Host IP address
  198.51.100.1         IPv4 address
  198.51.100.10        IPv4 address
```

Meaning When context-sensitive help is requested for the **hostip** value, Junos OS invokes the **hostip-expand.py** script. The script processes the user's input, if provided, and prints the valid completions in the CLI. If no user input is given, the script prints all possible values. When user input is provided, the script prints only matching values.

Configure a Telemetry Sensor in Junos

Using Junos telemetry streaming, you can turn any available state information into a telemetry sensor by means of the XML Proxy functionality. The NETCONF XML management protocol and Junos XML API fully document all options for every supported Junos OS operational request. After you configure XML proxy sensors, you can access data over NETCONF "get" remote procedure calls (RPCs).

This task shows you how to stream the output of a Junos OS operational mode command.

BEST PRACTICE: We recommend that you not use YANG files that map to an extensive or verbose Junos OS operational commands, such as **show interfaces** or **show route**. The use of such a file could result in very slow or no streaming of telemetry data or very high CPU usage for various processes.

This task requires the following:

- An MX Series, vMX Series, or PTX Series router operating Junos OS Release 17.3R2 or later.
- Installation of the required Network Agent package (network-agent-x86–32–17.4R1.16-C1.tgz or later).
- A telemetry data receiver, such as OpenNTI, to verify proper operation of your telemetry sensor.

In this task, you will stream the contents of the Junos OS command **show system users**.

show system users (vMX Series)

```
user@switch> show system users
```

```
USER     TTY      FROM                              LOGIN@  IDLE WHAT
user1   pts/0       172.31.12.36                    12:40PM     39 -cli (cli)

user2   pts/1       172,16.03.25                     3:01AM      - -cli (cli)
```

In addition to the expected list of currently logged-in users, the **show system users** output also provides the average system load as 1, 5 and 15 minutes. You can find the load averages by using the **show system users | display xml** command to view the XML tagging for the output fields. See **<load-average-1>**, **<load-average-5>**, and **<load-average-15>** in the XML tagging output below.

```
user@switch> show system users | display xml
```

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.4R1/junos">
    <system-users-information xmlns="http://xml.juniper.net/junos/17.4R1/junos">

        <uptime-information>
            <date-time junos:seconds="1520170982">1:43PM</date-time>
            <up-time junos:seconds="86460">1 day, 40 mins</up-time>
            <active-user-count junos:format="2 users">2</active-user-count>
            <load-average-1>0.70</load-average-1>
            <load-average-5>0.58</load-average-5>
            <load-average-15>0.55</load-average-15>
            <user-table>
                <user-entry>
                    <user>root</user>
                    <tty>pts/0</tty>
                    <from>172.21.0.1</from>
                    <login-time junos:seconds="1520167202">12:40PM</login-time>
                    <idle-time junos:seconds="0">-</idle-time>
                    <command>cli</command>
                </user-entry>
                <user-entry>
                    <user>mwiget</user>
                    <tty>pts/1</tty>
                    <from>66.129.241.10</from>
                    <login-time junos:seconds="1520170862">1:41PM</login-time>
                    <idle-time junos:seconds="60">1</idle-time>
                    <command>cli</command>
                </user-entry>
            </user-table>
```

```
        </uptime-information>
    </system-users-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

TIP: The uptime-information tag shown in the preceding output is a container that contains leafs, such as date-time, up-time, active-user-count. and load-average-1. Below is a sample YANG file for this container:

```
container uptime-information {
      dr:source "uptime-information"; // Exact name of the XML tag
      leaf date-time { // YANG model leaf
        type string; // Type of value
        dr:source date-time; // Exact name of the XML tag
      }
      leaf up-time { // YANG model leaf
        type string; // Type of value
        dr:source up-time; // Exact name of the XML tag
      }
      leaf active-user-count { // YANG model leaf
        type int32; // Type of value
        dr:source active-user-count; // Exact name of the XML tag
      }
      leaf load-average-1 { // YANG model leaf
        type string; // Type of value
        dr:source load-average-1; // Exact name of the XML tag
      }
      ...
```

TIP: The uptime-information tag also has another container named user-table that contains a list of user entries.

Below is a sample YANG file for this container:

```
container user-table { // "user-table" container which contains list of user-entry

        dr:source "user-table"; // Exact name of the XML tag
        list user-entry { // "user-entry" list which contains the users' details
 in form of leafs
          key "user"; // Key for the list "user-entry" which is a leaf in the
list "user-entry"
         dr:source "user-entry"; // Source of the list "user-entry" which is the
 exact name of the XML tag
          leaf user { // YANG model leaf
            dr:source user; // A leaf in the list "user-entry", exact name of the
 XML tag
            type string; // Type of value
          }
```

```
          leaf tty { // YANG model leaf
            dr:source tty; // A leaf in the list "user-entry", exact name of the
 XML tag
            type string; // Type of value
          }
          leaf from { // YANG model leaf
           dr:source from; // A leaf in the list "user-entry", exact name of the
 XML tag
            type string; // Type of value
          }
          leaf login-time { // YANG model leaf
            dr:source login-time; // A leaf in the list "user-entry", exact name
 of the XML tag
            type string; // Type of value
          }
          leaf idle-time { // YANG model leaf
            dr:source idle-time; // A leaf in the list "user-entry", exact name
of the XML tag
            type string; // Type of value
          }
          leaf command { // YANG model leaf
            dr:source command; // A leaf in the list "user-entry", exact name of
 the XML tag
            type string; // Type of value
          }
        }
      }
```


- Create a User-Defined YANG File on page 391
- Load the Yang File in Junos on page 394
- Collect Sensor Data on page 396
- Installing a User-Defined YANG File on page 398
- Troubleshoot Telemetry Sensors on page 399

Create a User-Defined YANG File

The YANG file defines the Junos CLI command to be executed, the resource path the sensors are placed under, and the key value pairs taken from the matching XML tags.

Custom YANG files for Junos OS conform to the YANG language syntax defined in RFC 6020 YANG 1.0 *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)* and RFC 7950 *The YANG 1.1 Data Modeling Language*. Certain directives need to be present in the file that configure XML proxy.

To use the **xmlproxyd** (daemon) process to translate telemetry data, create a ***render*.yang** file. In this file, the **dr:command-app** is set to **xmlproxyd**.

The XML proxy YANG filename and module name must start with **xmlproxyd_**:

- For the XML proxy YANG filename, add the extension **.yang**, for example, **xmlproxyd_sysusers.yang**
- For the module name, use the filename without the extension **.yang**, for example, **xmlproxyd_sysusers**

To simplify creating a YANG file, it's easiest to start by modifying a working example.

1. Provide a name for the module. The module name must start with **xmlproxyd_** and be the same name as the XML proxy YANG file name.

   For example, for an XML proxy YANG file called **sysusers.yang**, drop the **.yang** extension and name the module **xmlproxyd_sysusers**:

   module xmlproxyd_sysusers {

2. For the Junos Telemetry Interface, include the process (daemon) name **xmlproxyd**:

   dr:command-app "xmlproxyd";

3. Include the following RPC for the NETCONF get request:

   rpc juniper-netconf-get {

4. Specify the location of the output of the RPC, where *company-name* is the name you give to the location:

   dr:command-top-of-output "/company-name";

5. Include the following command to execute the RPC:

   dr:command-full-name "drend juniper-netconf-get";

6. Specify the CLI command from which to retrieve data. The Junos OS CLI command that gets executed at the requested sample frequency is defined under **dr:cli-command** and executed by the **xmlproxyd** daemon.

To retrieve command output for the Junos OS command **show system users**:

dr:cli-command "show system users";

7. Escalate privileges, logon as “root”, connect to the internal management socket via Telnet, and specify help for an RPC:

   dr: command-help “default <get> rpc”;

   When this is included in the YANG file, output that is helpful for debugging is displayed in the **help drend** output on the internal management socket:

```
telnet /var/run/xmlproxyd_mgmt
Trying /var/run/xmlproxyd_mgmt...
Connected to /var/run/xmlproxyd_mgmt.
Escape character is '^]'.
220 XMLPROXYD release 18.2I20180412_0904_bijchand built by bijchand on
2018-04-12 14:48:48 UTC
help drend
```

   200-juniper-netconf-get-0 system users <get> RPC

8. Specify the hierarchy and use the **dr:source** command to map to a container, a list, or a specific leaf. The absolute path under which the sensors will be reported is built from the output group **junos** plus **system-users-information**, concatenated by **/'**. The path **/junos/system-users-information/** is the path to query for information about this custom sensor.

   WARNING: You should not create a custom YANG model that conflicts or overlaps with predefined native paths (Juniper defined paths) and OpenConfig paths (resources). Doing so can result in undefined behavior.

   For example, do not create a model that defines new leafs at or augments nodes for resource paths such as **/junos/system/linecard/firewall**or **/interfaces**.

   A one-to-one mapping between container, leafs and the XML tag or value from the CLI command output is defined in the grouping referenced by **uses** within the output container. A *grouping* can be referred to multiple times in different container outputs. The container **system-users-information** below uses the grouping system-users-information. However, it is defined without the aforementioned one-to-one mapping for every container, list and leaf to an output XML tag from the CLI command XML output.

```
output {
  container junos {
    container system-users-information {
      dr:source "/system-users-information“;
      uses system-users-information-grouping;
```

```
    }
  }
}
```

9. The following YANG file shows how to include these commands to enable the **xmlproxyd** process to retrieve the full operational state and map it to the leafs in Juniper's own data model:

```
*/
```

```
/*
 * Example yang for generating OpenConfig equivalent of show system users
 */

module xmlproxyd_sysusers {
  yang-version 1;

  namespace "http://juniper.net/yang/software";

  import drend {
    prefix dr;
  }

  grouping system-users-information-grouping {
    container uptime-information {
      dr:source "uptime-information";
      leaf date-time {
        type string;
        dr:source date-time;
      }
      leaf up-time {
        type string;
        dr:source up-time;
      }
      leaf active-user-count {
        type int32;
        dr:source active-user-count;
      }
      leaf load-average-1 {
        type string;
        dr:source load-average-1;
      }
      leaf load-average-5 {
        type string;
        dr:source load-average-5;
      }
      leaf load-average-15 {
        type string;
        dr:source load-average-15;
      }
      container user-table {
        dr:source "user-table";
        list user-entry {
          key "user";
          dr:source "user-entry";
          leaf user {
            dr:source user;
```

```
          type string;
        }
        leaf tty {
          dr:source tty;
          type string;
        }
        leaf from {
          dr:source from;
          type string;
        }
        leaf login-time {
          dr:source login-time;
          type string;
        }
        leaf idle-time {
          dr:source idle-time;
          type string;
        }
        leaf command {
          dr:source command;
          type string;
        }
      }
    }
  }
}

dr:command-app "xmlproxyd";
rpc juniper-netconf-get {
  dr:command-top-of-output "/company-name";
  dr:command-full-name "drend juniper-netconf-get";
  dr:cli-command "show system users";
  dr:command-help "default <get> rpc";
```

```
output {
      container company-name {
        container system-users-information {
          dr:source "/system-users-information";
          uses system-users-information-grouping;
        }
      }
    }
  }
}
```

Load the Yang File in Junos

After the YANG file is complete, upload the YANG file and verify that the module is created.

1. Upload the YANG file to the router.

2. Register the YANG file using the **request system yang add package** command.

   ```
   user@switch> request system yang add package sysusers proxy-xml module
   xmlproxyd_sysusers.yang
   ```

```
XML proxy YANG module validation for xmlproxyd_sysusers.yang : START
XML proxy YANG module validation for xmlproxyd_sysusers.yang : SUCCESS
JSON generation for xmlproxyd_sysusers.yang : START
JSON generation for xmlproxyd_sysusers.yang: SUCCESS
```

NOTE: Starting in Junos OS Release 18.3R1, adding, deleting, or updating YANG packages in configuration mode with the run command is not supported.

3. Verify that the module (sensor) is registered using the **show system yang package sysusers** command, where **sysusers** is the name of the package:

```
user@switch> show system yang package sysusers
```

```
Package ID               :sysusers
XML Proxy YANG Module(s) :xmlproxyd_sysusers.yang
```

4. Enable gRPC in the Junos OS configuration:

```
user@switch> set system services extension-service request-response grpc port 32767
```

Collect Sensor Data

Use your favorite collector to pull the newly created telemetry sensor data from the device. .The following instructions use the collector *jtimon*. For information about jtimon setup, see Junos Telemetry Interface client.

1. Create a simple configuration file, here named **vmx1.json**. Adjust the host IP address and the port, as needed. The path **/junos/system-users-information** is specified. The **freq** field is defined in MicroSoft, streaming a new set of key value pairs every 5 seconds. Optionally, you can add multiple paths.

```
$ cat vmx1.json
{
  "host": "172.16.122.182
  "port": 32767
  "cid": "my-client-id",
  "grpc" : {
    "ws" : 524289
  },
  "paths": {
     {
       "path": "/junos/system-users-information/",
       "freq": 5000
     },
     {
       "path": "/junos/additional-path/",  <-OPTIONAL
       "freq": 5000
     }
   }
}
```

2. Launch the collector, using either your own compiled file or an automatically built image from Docker Hub.
   The sample query output below shows the sensor report by path. Every key is sent in human-readable form as an absolute path. In case of lists, the absolute path contains an index in the form of XPATH which is ideal to group values from a (time series) database, such as InfluxDB. For example, the output below shows the path **/junos/system-users-information/uptime-information/user-table/user-entry[user='ab']/**.

   You can terminate the stream of sensor data using Ctrl-C.

   $ docker run -tu --rm -v $(PWD):/u mw/jtimon --config vmx1.json --print

```
gRPC headers from Junos:
  init-response: [response { subscription_id 1} path_list {path:
"junos/system-users-information/" sample-frequency: 5000 } ]
  content-type: [application/grpc]
  grpc-accept-encoding: [identity,deflate,gzip]
2018/03/04 17:13:19 system-id vmxdockerlight_vmx1_1
2018/03/04 17:13:19 component_id 65535
2018/03/04 17:13:19 sub_component_id: 0
2018/03/04 17:13:19 path:
sensor_1000:/junos/system-users-information/:/junos/system-users-information/
2018/03/04 17:13:19 sequence_number: 16689
2018/03/04 17:13:19 timestamp: 1520183589391
```

```
2018/03/04 17:13:19 sync_response: %!d(bool=false)
2018/03/04 17:13:19 key: __timestamp__
2018/03/04 17:13:19 uint_value: 1520183589391
2018/03/04 17:13:19 key: __junos_re_stream_creation_timestamp--
2018/03/04 17:13:19 uint value: 1520183589372
2018/03/04 17:13:19 key: __junos_re_payload-get_timestamp__
2018/03/04 17:13:19 uint_value: 1520183589390
2018/03/04 17:13:19 key:
/junos/system-users-information/uptime-information/date-time
2018/03/04 17:13:19 str-value: 5:13PM
2018/03/04 17:13:19 key:
/junos/system-users-inforamtion/uptime-information/up-time
2018/03/04 17:13:19 str-value: 1 day, 4:10
2018/03/04 17:13:19 key:
/junos/system-users-information/uptime-information/active-user-count
2018/03/04 17:13:19 int_value: 2
2018/03/04 17:13:19 key:
/junos/system-users-inforamtion/uptime-information/load-average-1
2018/03/04 17:13:19 str_value: 0.62
2018/03/04 17:13:19 key:
/junos/system-users-information/uptime-information/load-average-5
2018/03/04 17:13:19 str_value: 0.56
2018/03/04 17:13:19 key:
/junos/system-users-inforamtion/uptime-information/load-average-15
2018/03/04 17:13:19 str_value: 0.53
2018/03/04 17:13:19 key: __prefix__
2018/03/04 17:13:19 str_value:
/junos/system-users-information/uptime-information/user-table/user-entry[user='ab']/
2018/03/04 17:13:19 key: tty
2018/03/04 17:13:19 str_value: pts/1
2018/03/04 17:13:19 key: from
2018/03/04 17:13:19 str-value: 172,16.04.25
2018/03/04 17:13:19 key: login-time
2018/03/04 17:13:19 str_value: 5:12PM
2018/03/04 17:13:19 key: idle-time
2018/03/04 17:13:19 str-value: -
2018/03/04 17:13:19 key: command
2018/03/04 17:13:19 str_value: -cl
2018/03/04 17:13:19 system_id: vmxdockerlight_vmx1_1
2018/03/04 17:13:19 component_id: 65535
2018/03/04 17:13:19 sub_component_id: 0
2018/03/04 17:13:19 <output truncated>
```

The sample query shown below shows two sensor reports per path, then I terminated it with Ctrl-C. Every key is sent in human readable form as an absolute path and in case of lists, contains an index in form of XPATH, ideal to group values from a (time series) database like InfluxDB e.g. /junos/system-users-information/uptime-information/user-table/user-entry[user='ab']/

3. Verify that the module (sensor) is loaded using the **show system yang package sysusers** command, where **sysusers** is the name of the package:

```
user@switch> show system yang package sysusers
```

```
Package ID               :sysusers
XML Proxy YANG Module(s) :xmlproxyd_sysusers.yang
```

4. Enable gRPC in the Junos OS configuration:

```
user@switch> set system services extension-service request-response grpc port 32767
```

Installing a User-Defined YANG File

To add, validate, modify, or delete a user-defined YANG file for XML proxy for the Junos Telemetry Interface, use the **request system yang** set of commands from the operational mode:

1. Specify the name of the XML proxy YANG file and the file path to install it. This command creates a **.json** file in the **/opt/lib/render** directory.

```
user@switch> request system yang add package package-name proxy-xml module
  file-path-name
```

NOTE: This command can be performed only on the current routing engine.

To add multiple YANG modules with the **request system yang add package** *package-name* **proxy-xml module** command, enclose the *file-path-name* in brackets: **[** *file-path-name 1 file-path-name 2* **]**

2. (Optional) Validate an module before adding it to the router using the **request system yang validate proxy-xml module** *module-name* command. .

```
user@switch> request system yang validate proxy-xml module module-name
```

The output **XML proxy YANG module validation for xmlproxyd_<module-name> : SUCCESS** indicates successful module validation.

Mismatch error sometimes occur. If the command returns the error below, you can eliminate the error by using Junos OS Release 17.3R2 or later:

```
user@switch> request system yang validate proxy-xml module
  xmlproxyd_sysusers.yang
  error: illegal identifier <identifier> , must not start with [xX][mM][lL]
```

3. (Optional) Update an existing XML proxy YANG file that was previously added.

```
user@switch> request system yang update package-name proxy-xml module
  file-path-name
```

4. Delete an existing XML proxy YANG file.

```
user@switch> request system yang delete package-name
```

5. Verify that the YANG file has been installed by entering the show system yang package command.

```
user@switch> show system yang package package-name
```

See Also

- Understanding YANG on Devices Running Junos OS on page 277
- *Installing the Network Agent Package (Junos Telemetry Interface)*
- *Guidelines for gRPC Sensors (Junos Telemetry Interface)*
- Sending Requests to the NETCONF Server on page 50

Troubleshoot Telemetry Sensors

Problem **Description:** Use the following methods to troubleshoot user-define telemetry sensors:

- Execute a tcpdump for the interface your gRPC requests came from (for this task, interface **fxp0** was used).

```
user@switch>monitor traffic interface fxp0 no-resolve matching "tcp port 32767"
```

- Enable traceoptions using the **set services analytics traceoptions flag xmlproxy** command. Check the **xmlproxyd** log file for confirmation of whether the CLI command's RPC was sent and if a response was received:

1. Issue the **show log xmlproxyd** command to show the xmlproxyd log. The value for the field **xmlproxy_execute_cli_command:** indicates if the RPC was sent or not. The value for the field **xmlproxy_build_context** indicates the command.

```
user@switch>show log xmlproxyd
Mar 4 18:52:46 vmxdockerlight_vmx1_1 clear-log[52495]: logfile cleared
Mar  4 18:52:51 xmlproxy_telemetry_start_streaming: sensor
/junos/system-users-information/
Mar  4 18:52:51 xmlproxy_build_context: command show system users merge-tag:
Mar  4 18:52:51 <command format="xml">show system users</command>
Mar  4 18:52:51 xmlproxy_execute_cli_command: Sent RPC..
Mar  4 18:52:51 <system-users-information
xmlns="http://xml.juniper.net/junos/17.4R1/junos"
xmlns:junos="http://xml.juniper.net/junos/*/junos">
<uptime-information>
<date-time junos:seconds="1520189571">
6:52PM
</date-time>
<up-time junos:seconds="107400">
1 day,  5:50
</up-time>
<active-user-count junos:format="1 users">
1
</active-user-count>
<load-average-1>
0.94
</load-average-1>
<load-average-5>
```

```
0.73
</load-average-5>
<load-average-15>
0.65
```

See Also

- Understanding YANG on Devices Running Junos OS on page 277
- *Installing the Network Agent Package (Junos Telemetry Interface)*
- *Configurable NETCONF Proxy for Junos Telemetry Interface*
- *Guidelines for gRPC Sensors (Junos Telemetry Interface)*
- Sending Requests to the NETCONF Server on page 50

PART 7

OpenDaylight Integration

CHAPTER 20

Configuring OpenDaylight Integration

Configuring Interoperability Between MX Series Routers and OpenDaylight

OpenDaylight (ODL), hosted by the Linux Foundation, is an open source platform for network programmability aimed at enhancing software-defined networking (SDN).

Starting from the Junos OS Release 17.3R1, you can configure interoperability between MX Series routers and the ODL controller. ODL provides a southbound Network Configuration Protocol (NETCONF) connector API, which uses NETCONF and YANG models, to interact with a network device. A southbound interface, an OpenFlow (or alternative) protocol specification, enables communication between ODL and routers or switches. Once this feature is implemented on an MX Series router, you can use the ODL platform to carry out any configuration change in the routers, orchestrate and provision the router, and execute Remote Procedure Calls (RPCs) to the router to get state information.

Setting up interoperability between ODL and an MX Series router involves the following tasks:

Configuring NETCONF on the MX Series Router

As a prerequisite for configuring interoperability between ODL and MX Series router, you must configure NETCONF on the router. NETCONF is used by the ODL controller to interact with southbound devices.

To configure NETCONF in the router:

1. Enable access to the NETCONF SSH subsystem.

   [edit]

   user@host# set system services netconf ssh

2. Set the NETCONF server to enforce certain behaviors that are compliant with RFC 4741, NETCONF Configuration Protocol, during NETCONF sessions.

   [edit]

   user@host# set system services netconf rfc-compliant

3. Use the **yang-compliant** configure the system to prevent exporting any hidden or unsupported configuration hierarchies in get-config RPC output. This output gives the details of current configuration on the network device.

   [edit]

   user@host# set system services netconf yang-compliant

4. Commit the changes.

   [edit]

   user@host# commit

Configuring NETCONF Trace Options

After you configure NETCONF on the router, you need to configure NETCONF Trace options, by using the traceoptions statement. To know more about NETCONF and Junos XML protocol tracing operations, read this article.

To configure NETCONF trace options:

1. Configure the details of the file to receive the output of the tracing operation. You can configure the file name, maximum file size, and flags to indicate tracing operations, by using the following statements:

   [edit]

   user@host# set system services netconf traceoptions file *file name*

   user@host# set system services netconf traceoptions file size *size*

   user@host# set system services netconf traceoptions flag *flag*

   user@host# commit

To know more on how to configure tracing operations for NETCONF and Junos XML protocol sessions, see this example.

Connecting ODL to MX Series Router

After NETCONF is configured on the MX Series router, you need to connect the ODL controller to the router to complete the process. For more details on this, see this ODL documentation.

Related Documentation

- NETCONF and Junos XML Protocol Tracing Operations Overview on page 71
- NETCONF Session Overview on page 29

PART 8

Configuration Statements and Operational Commands

CHAPTER 21

Configuration Statements (Ephemeral Configuration Database)

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.
Statement introduced in Junos OS Release 18.3R1 on ACX Series and PTX Series routers and QFX Series 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 multiple 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 182. 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. If you commit invalid configuration data to the database, it can cause Junos OS processes to restart or even crash and result in disruption to the system or network.

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 177
- Enabling and Configuring Instances of the Ephemeral Configuration Database on page 184
- Committing an Instance of the Ephemeral Configuration Database Using the NETCONF or Junos XML Protocol on page 191

instance (Ephemeral Database)

Syntax instance *instance-name*;

Hierarchy Level [edit system configuration-database ephemeral]

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.
Statement introduced in Junos OS Release 18.3R1 on ACX Series and PTX Series routers and QFX Series switches.

Description Enable an instance of the ephemeral configuration database.

Starting in Junos OS Release 18.2R1, you can configure up to seven user-defined instances of the ephemeral configuration database. In earlier releases, you can configure up to eight user-defined instances. 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.

NOTE: **When you configure an ephemeral instance, you can specify its placement in the configuration by using the insert command instead of the set command.**

Options ***instance-name***—User-defined name for an instance of the ephemeral configuration database.

The instance name must contain only alphanumeric characters, hyphens, and underscores, and it must not exceed 32 characters in length. In addition, starting in Junos OS Release 17.1R3, 17.2R3, 17.3R3, 17.4R2, and 18.1R1, the name of an user-defined instance cannot be **default**.

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

Related Documentation
- Enabling and Configuring Instances of the Ephemeral Configuration Database on page 184

- Example: Configuring the Ephemeral Configuration Database Using NETCONF on page 194
- Understanding the Ephemeral Configuration Database on page 177

CHAPTER 22

Configuration Statements (NETCONF)

connection-limit

Syntax

```
connection-limit limit;
```

Hierarchy Level

```
[edit system services finger],
[edit system services ftp],
[edit system services netconf ssh],
[edit system services ssh],
[edit system services telnet],
[edit system services xnm-clear-text],
[edit system services xnm-ssl]
```

Release Information

Statement introduced before Junos OS Release 7.4.
Statement introduced in Junos OS Release 9.0 for EX Series switches.
Statement introduced in Junos OS Release 11.1 for the QFX Series.
Statement introduced in Junos OS Release 14.1X53-D20 for OCX Series switches.

Description

Configure the maximum number of connections sessions for each type of system services (finger, ftp, ssh, telnet, xnm-clear-text, or xnm-ssl) per protocol (either IPv6 or IPv4).

Options

limit—(Optional) Maximum number of established connections per protocol (either IPv6 or IPv4).

Range: 1 through 250

Default: 75

NOTE: The actual number of maximum connections depends on the availability of system resources, and might be fewer than the configured connection-limit value if the system resources are limited.

Required Privilege Level

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

Related Documentation

- *Configuring clear-text or SSL Service for Junos XML Protocol Client Applications*
- *Configuring DTCP-over-SSH Service for the Flow-Tap Application*
- *Configuring Finger Service for Remote Access to the Router*
- *Configuring FTP Service for Remote Access to the Router or Switch*
- *Configuring SSH Service for Remote Access to the Router or Switch*
- *Configuring Telnet Service for Remote Access to a Router or Switch*

netconf

Syntax

```
netconf {
  rfc-compliant;
  ssh {
    connection-limit limit;
    port port;
    rate-limit limit;
  }
  traceoptions {
    file <filename> <files number> <match regular-expression> <size size>
      <world-readable | no-world-readable>;
    flag flag;
    no-remote-trace;
    on-demand;
  }
  yang-compliant;
  yang-modules {
    device-specific;
    emit-extensions;
  }
}
```

Hierarchy Level

```
[edit system services]
```

Release Information

Statement introduced in Junos OS Release 7.5.

Description

Configure the NETCONF XML management protocol.

The remaining statements are explained separately. See CLI Explorer.

Default

If you do not include the **netconf** statement, NETCONF connections are not permitted.

Required Privilege Level

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

Related Documentation

- connection-limit on page 416
- port (NETCONF) on page 418
- rate-limit on page 419
- ssh (NETCONF) on page 421
- traceoptions (NETCONF and Junos XML Protocol) on page 422

port (NETCONF)

Syntax port *port-number*;

Hierarchy Level [edit system services netconf]

Release Information Statement introduced in Junos OS Release 10.0.

Description Configure the TCP port used for NETCONF-over-SSH connections.

NOTE:

- The configured port accepts only NETCONF-over-SSH connections. Regular SSH session requests for this port are rejected.
- The default SSH port (22) continues to accept NETCONF sessions even with a configured NETCONF server port. To disable the SSH port from accepting NETCONF sessions, you can specify this in the login event script.
- We do not recommend configuring the default ports for FTP (21) and Telnet (23) services for configuring NETCONF-over-SSH connections.

Options port *port-number*—Port number on which to enable incoming NETCONF connections over SSH.

Default: 830 (as specified in RFC 4742, *Using the NETCONF Configuration Protocol over Secure Shell (SSH)*)

Range: 1 through 65535

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

Related Documentation

- *NETCONF XML Management Protocol Guide*
- *Configuring NETCONF-Over-SSH Connections on a Specified TCP Port*

rate-limit

Syntax

```
rate-limit limit;
```

Hierarchy Level

```
[edit system services finger],
[edit system services ftp],
[edit system services netconf ssh],
[edit system services ssh],
[edit system services telnet],
[edit system services tftp-server],
[edit system services xnm-clear-text],
[edit system services xnm-ssl]
```

Release Information

Statement introduced before Junos OS Release 7.4.
Statement introduced in Junos OS Release 9.0 for EX Series switches.
Statement introduced in Junos OS Release 14.1X53-D20 for OCX Series switches.
Statement introduced in Junos OS Release 11.1 for the QFX Series.

Description

Configure the maximum number of connections attempts per minute, per protocol (either IPv6 or IPv4) on an access service. For example, a rate limit of 10 allows 10 IPv6 telnet session connection attempts per minute and 10 IPv4 telnet session connection attempts per minute.

Default

150 connections

Options

rate-limit *limit*—(Optional) Maximum number of connection attempts allowed per minute, per IP protocol (either IPv4 or IPv6).

Range: 1 through 250

Default: 150

Required Privilege Level

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

Related Documentation

- *Configuring clear-text or SSL Service for Junos XML Protocol Client Applications*

rfc-compliant (NETCONF)

Syntax rfc-compliant;

Hierarchy Level [edit system services netconf]

Release Information Statement introduced in Junos OS Release 15.1.

Description Require that the NETCONF server enforce certain behaviors that are compliant with RFC 4741, *NETCONF Configuration Protocol*, during NETCONF sessions.

When you configure the **rfc-compliant** statement:

- The NETCONF server explicitly declares the NETCONF namespace in its replies and qualifies all NETCONF tags with the **nc** prefix.
- **<get>** and **<get-config>** operations that return no configuration data do not include an empty **<configuration>** element in RPC replies.
- On devices running Junos OS Release 17.2R1 or later, the NETCONF server sets the default namespace for the **<configuration>** element to the same namespace as in the corresponding YANG model.
- On devices running Junos OS Release 18.4R1 or later, the NETCONF server omits **<rpc-error>** elements with a severity level of warning in its replies when the operation is successful and returns an **<ok/>** element.

Default If you do not include the **rfc-compliant** statement:

- The NETCONF server sets the default namespace to the NETCONF namespace in RPC replies.
- **<get>** and **<get-config>** operations that return no configuration data include an empty **<configuration>** element in RPC replies.
- The NETCONF server does not set the default namespace for the **<configuration>** element to the same namespace as in the corresponding YANG model.
- The NETCONF server might issue an RPC reply that includes both an **<rpc-error>** element with a severity level of warning and an **<ok/>** element.

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

Related Documentation
- Configuring RFC-Compliant NETCONF Sessions on page 66

ssh (NETCONF)

Syntax

```
ssh {
  connection-limit limit;
  port port-number;
  rate-limit limit;
}
```

Hierarchy Level

[edit system services netconf]

Description

Enable access to the NETCONF SSH subsystem using the default port number 830, as specified by RFC 4742.

Options

The remaining statements are explained separately. See CLI Explorer.

Required Privilege Level

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

Related Documentation

- connection-limit on page 416
- netconf on page 417
- port (NETCONF) on page 418
- rate-limit on page 419

traceoptions (NETCONF and Junos XML Protocol)

Syntax

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

Hierarchy Level

```
[edit system services netconf]
```

Release Information

Statement introduced in Junos OS Release 12.2.
Support for Junos XML protocol sessions added in Junos OS Release 16.1.
Option **flag debug** introduced in Junos OS Release 16.1.

Description

Define 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.

Default

If you do not include this statement, NETCONF and Junos XML protocol-specific tracing operations are not performed.

Options

file ***filename***—Name of the file to receive the output of the tracing operation. All files are placed in the **/var/log** directory.

Default: **/var/log/netconf**

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

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

Range: 2 through 1000 files

Default: 10 files

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, ***size*k** to specify KB, ***size*m** to specify MB, or ***size*g** 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

- NETCONF and Junos XML Protocol Tracing Operations Overview on page 71
- Example: Tracing NETCONF and Junos XML Protocol Session Operations on page 72
- netconf on page 417

CHAPTER 23

Configuration Statements (Translation Scripts)

max-datasize

Syntax max-datasize *size*;

Hierarchy Level
[edit event-options event-script],
[edit system extension extension-service application],
[edit system scripts commit],
[edit system scripts op],
[edit system scripts snmp],
[edit system scripts translation]

Release Information
Statement introduced in Junos OS Release 12.3.
Statement introduced in Junos OS Release 13.2X51-D10 for QFX Series switches.
Support at the **[edit system extension extension-service application]** hierarchy level introduced in Junos OS Release 16.1 for MX80, MX104, MX240, MX480, MX960, MX2010, MX2020, and vMX.
Support at the **[edit system scripts translation]** hierarchy level introduced in Junos OS Release 16.1.

Description Maximum amount of memory allocated for the data segment during execution of a script of the configured type. Junos OS sets the maximum memory limit for the executing script to the configured value irrespective of the total memory available on the system at the time of execution. If the executing script exceeds the specified maximum memory limit for that script type, it exits gracefully.

NOTE: For op scripts, the max-datasize statement is only enforced for op scripts that are local to the device. If you execute an op script from a remote location using the op url command, Junos OS uses the default memory allocation settings.

NOTE: For op scripts run with the max-datasize statement configured for the minimum, an error is thrown. In classic Junos OS, the error is "Memory allocation failed." In Junos OS Evolved, the error is "Out of memory."

Default If you do not include the **max-datasize** statement, the system allocates half of the total available memory of the system up to a maximum value of 128 MB for the data segment portion of the executed script.

Options ***size***—Maximum amount of memory allocated for the data segment during execution of a script of the given type. If you do not specify a unit of measure, the default is bytes.

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

Range:

- 32-bit Junos OS—23,068,672 bytes (22 MB) through 1,073,741,824 bytes (1 GB)
- 64-bit Junos OS—23,068,672 bytes (22 MB) through 1,073,741,824 bytes (1 GB) (SNMP scripts)
- 64-bit Junos OS—23,068,672 bytes (22 MB) through 3,221,225,472 bytes (3 GB) (commit, event, op, translation, and extension service scripts)

NOTE: The maximum memory for extension service scripts in 64-bit Junos OS images is 3,221,225,472 bytes (3 GB) starting in Junos OS Releases 16.1R4, 16.2R2, and 17.1R1. Prior to these releases, the maximum is 1,073,741,824 bytes (1 GB).

Required Privilege Level

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

Related Documentation

- *max-policies*
- *Understanding Limits on Executed Event Policies and Memory Allocation for Scripts*
- *Example: Configuring Limits on Executed Event Policies and Memory Allocation for Scripts*

translation

Syntax

```
translation {
  max-datasize;
}
```

Hierarchy Level [edit system scripts]

Release Information Statement introduced in Junos OS Release 16.1.

Description Configure options for translation scripts.

Options The remaining statements are explained separately. See CLI Explorer.

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

Related Documentation

- *Storing and Enabling Scripts*

CHAPTER 24

Configuration Statements (YANG)

yang-compliant (NETCONF)

Syntax

```
yang-compliant;
```

Hierarchy Level

```
[edit system services netconf]
```

Release Information Statement introduced in Junos OS Release 17.3.

Description Prevent hidden or unsupported configuration hierarchies in Junos from being exported in **get-config** RPC outputs. Users cannot configure the hidden statements from the ODL controller.

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

Related Documentation

- rfc-compliant on page 420
- ssh on page 421

yang-modules (NETCONF)

Syntax

```
yang-modules {
  device-specific;
  emit-extensions;
}
```

Hierarchy Level

[edit system services netconf]

Release Information

Statement introduced in Junos OS Release 17.4R1.

Description

Configure how the device running Junos OS serves the native YANG modules.

Default

If you do not include the **yang-modules** statement, the device running Junos OS serves the family-specific YANG data models that are shipped with the device.

Options

device-specific—Instruct the device to serve device-specific YANG data models instead of the family-specific YANG data models that are shipped with the device.

emit-extensions—Instruct the device to serve YANG data models that explicitly include Junos OS extension statements.

Required Privilege Level

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

Related Documentation

- netconf on page 417

CHAPTER 25

Operational Commands (Ephemeral Configuration Database)

- show ephemeral-configuration

show ephemeral-configuration

List of Syntax

Syntax (Junos OS Release 18.2R1 and Later) on page 432
Syntax (Junos OS Release 18.1 and Earlier) on page 432

Syntax (Junos OS Release 18.2R1 and Later)

```
show ephemeral-configuration (instance instance-name | merge)
```

Syntax (Junos OS Release 18.1 and Earlier)

```
show ephemeral-configuration
<instance-name>
```

Release Information

Command introduced in Junos OS Release 16.2R2 on MX Series and T Series routers.
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.
Command introduced in Junos OS Release 18.1R1 on EX2300, EX3400, EX4300, EX4600, and EX9200 switches.
instance and **merge** options added in Junos OS Release 18.2R1.
Command introduced in Junos OS Release 18.3R1 on ACX Series and PTX Series routers and QFX Series switches.

Description

Display configuration data committed to the ephemeral configuration database.

Options

none—In Junos OS Release 18.1 and earlier, display the configuration committed to the default instance of the ephemeral configuration database.

instance-name—(Optional) Name of a user-defined ephemeral instance for which to display the committed ephemeral configuration data.

instance *instance-name*—Display the configuration committed to an instance of the ephemeral configuration database.

- To display the configuration data in the default ephemeral instance, set the instance name to **default**.
- To display the configuration data for sensors that have been provisioned by an external collector to export data through gRPC, set the instance name to **junos-analytics**.
- To display the configuration data in a user-defined instance, specify the name of an instance configured at the **[edit system configuration-database ephemeral instance]** hierarchy level.

merge—Display the configuration data in all instances of the ephemeral configuration database merged with the complete post-inheritance view of the static configuration database.

NOTE: In Junos OS Release 18.1 and earlier, to display the configuration data in all instances of the ephemeral configuration database merged with the complete post-inheritance view of the static configuration database, use the **show ephemeral-configuration | display merge** command.

Required Privilege Level view

Related Documentation
- Enabling and Configuring Instances of the Ephemeral Configuration Database on page 184

List of Sample Output

Sample Output

show ephemeral-configuration (Junos OS Release 18.1 or earlier)

```
user@host> show ephemeral-configuration
## Last changed: 2017-02-12 17:15:48 PDT
protocols {
    mpls {
        label-switched-path to-cust1 {
            to 198.51.100.1;
        }
    }
}
```

show ephemeral-configuration eph1 (Junos OS Release 18.1 or earlier)

```
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;
        }
    }
}
```

show ephemeral-configuration instance eph1 (Junos OS Release 18.2R1 or later)

```
user@host> show ephemeral-configuration instance eph1
## Last changed: 2017-02-10 13:20:32 PDT
protocols {
```

```
    mpls {
        label-switched-path to-hastings {
            to 192.0.2.1;
        }
    }
}
```

CHAPTER 26

Operational Commands (YANG)

request system yang add

Syntax

```
request system yang add package package-name <proxy-xml> module [modules]
<action-script [scripts]>
<translation-script  [scripts]>
<deviation-module [modules]>
<snmp>
```

Release Information

Command introduced in Junos OS Release 16.1R1 on MX Series and T Series routers.
Command introduced in Junos OS Release 17.1R1 on EX Series and QFX Series switches and PTX Series routers.
Command introduced in Junos OS Release 17.3R1 on SRX345, SRX1500, SRX4100, SRX4200, SRX5400, SRX5600, and SRX5800 devices and vSRX instances.
proxy-xml option introduced in Junos OS Release 17.3R1 on MX Series and PTX Series routers.
Command introduced in Junos OS Release 18.1R1 on ACX Series routers.
snmp option introduced in Junos OS Release 18.3R1.

Description

Define a new YANG package with the modules, deviation modules, and scripts that are added to the device as part of the package, and merge the data models defined in the modules with the Junos OS schema. When you add a custom YANG data model to the device, you must also add at least one translation script or one action script, which provides the mapping between the new data model and Junos OS. To add multiple modules or scripts, include a space-delimited list of absolute or relative file paths enclosed in brackets.

NOTE: To install OpenConfig modules that are packaged as a compressed tar file, use the **request system software add** command. OpenConfig modules and scripts that are installed using the **request system software add** command are always associated with the package identifier **openconfig**.

When you create a new package, the device stores copies of the module and script files in a new location. The device also stores copies of the action script and translation script files under the **/var/db/scripts/action** and **/var/db/scripts/translation** directories, respectively. Junos OS validates the syntax of the modules and scripts, rebuilds its schema to include the new data models, and then validates the active configuration against this schema. Newly added RPCs and configuration hierarchies are immediately available for use.

NOTE: Devices that use the ephemeral configuration database will delete all ephemeral configuration data in the process of rebuilding the schema.

NOTE: To prevent CLI-related or configuration database errors, we recommend that you do not perform any CLI operations or change the configuration while a device is in the process of adding, updating, or deleting a YANG package and modifying the schema.

NOTE: Starting in Junos OS Release 18.3R1, adding, deleting, or updating YANG packages in configuration mode with the run command is not supported.

Options

action-script [*scripts*]—List of paths for one or more action scripts to add to the device as part of the package.

module [*modules*]—List of paths for one or more YANG modules to add to the device as part of the package. The device merges the data models defined in the modules with the Junos OS schema.

deviation-module [*modules*]—(Optional) List of paths for one or more modules that define deviation statements that should be applied to modules in the package.

package *package-name*—User-defined identifier that represents the collection of YANG modules and scripts.

proxy-xml module [*modules*]—List of paths for one or more new modules that provide user-defined OpenConfig mappings for the XML Proxy process to translate Junos Telemetry Interface statistics exported through gRPC into key-value pairs.

snmp —List of paths for one or more YANG modules to copy to a predefined location and convert it to JSON format. Later snmpd parses this JSON file and builds its internal database. Requires the **package *package-name*** option.

translation-script [*scripts*]—List of paths for one or more translation scripts to add to the device as part of the package.

Required Privilege Level

maintenance

Related Documentation

- Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS on page 299
- Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS on page 297
- Configure a Telemetry Sensor in Junos on page 387
- request system yang update on page 445

- show system yang package on page 453
- *Customized SNMP MIBs for Syslog Traps*

Sample Output

request system yang add

```
user@host> request system yang add package p1 module [yang/if.yang yang/if-aggregate.yang
yang/if-show.yang] deviation-module yang/deviation/if-devs.yang
translation-script translation/if.slax action-script action/if-show.py

YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
script check succeeds
Scripts syntax validation : SUCCESS
Scripts syntax validation : START
Scripts syntax validation : SUCCESS
TLV generation: START
TLV generation: SUCCESS
Building schema and reloading /config/juniper.conf.gz ...
Activating /config/juniper.conf.gz ...
mgd: commit complete
Restarting mgd ...

WARNING: cli has been replaced by an updated version:
CLI release 16.1R1 built by builder on 2016-03-30 13:46:11 UTC
Restart cli using the new version ? [yes,no] (yes) yes

Restarting cli ...
user@host>
```

request system yang delete

Syntax **request system yang delete** *package-name*

Release Information Command introduced in Junos OS Release 16.1R1 on MX Series and T Series routers.
Command introduced in Junos OS Release 17.1R1 on EX Series and QFX Series switches and PTX Series routers.
Command introduced in Junos OS Release 17.3R1 on SRX345, SRX1500, SRX4100, SRX4200, SRX5400, SRX5600, and SRX5800 devices and vSRX instances.
Command introduced in Junos OS Release 18.1R1 on ACX Series routers.

Description Remove the given YANG package and all of its modules and scripts from the device, and remove the data models associated with that package from the Junos OS schema.

CAUTION: Before you delete a YANG package, ensure that the active configuration does not contain configuration data that has dependencies on the data models added by that package.

NOTE: You must use the **request system software delete** command to remove OpenConfig packages that were installed from a compressed tar file using the **request system software add** command.

When you delete a package, Junos OS rebuilds its schema to remove the data models associated with that package and then validates the active configuration against the newly updated schema. The device removes the copies of the module and script files that were generated when the package was created. The device also removes the copies of the package's action script and translation script files that are stored under the **/var/db/scripts/action** and **/var/db/scripts/translation** directories. If you downloaded the original module and script files to a different location, the original files remain unchanged.

NOTE: Devices that use the ephemeral configuration database will delete all ephemeral configuration data in the process of rebuilding the schema.

NOTE: To prevent CLI-related or configuration database errors, we recommend that you do not perform any CLI operations or change the configuration while a device is in the process of adding, updating, or deleting a YANG package and modifying the schema.

NOTE: Starting in Junos OS Release 18.3R1, adding, deleting, or updating YANG packages in configuration mode with the run command is not supported.

Options *package-name*—Name of the YANG package to remove.

Required Privilege Level maintenance

Related Documentation

- Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS on page 299
- Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS on page 297
- request system yang add on page 436
- show system yang package on page 453

Sample Output

request system yang delete

```
user@host> request system yang delete p1
Building schema and reloading /config/juniper.conf.gz ...
Activating /config/juniper.conf.gz ...
mgd: commit complete
Restarting mgd ...

WARNING: cli has been replaced by an updated version:
CLI release 16.1R1 built by builder on 2016-03-30 13:46:11 UTC

Restart cli using the new version ? [yes,no] (yes) yes

Restarting cli ...
```

request system yang disable

Syntax **request system yang disable *package-name***

Release Information Command introduced in Junos OS Release 16.1R1 on MX Series and T Series routers.
Command introduced in Junos OS Release 17.1R1 on EX Series and QFX Series switches and PTX Series routers.
Command introduced in Junos OS Release 17.3R1 on SRX345, SRX1500, SRX4100, SRX4200, SRX5400, SRX5600, and SRX5800 devices and vSRX instances.
Command introduced in Junos OS Release 18.1R1 on ACX Series routers.

Description Disable the translation scripts associated with the given YANG package.

Translation scripts convert configuration data corresponding to YANG data models into Junos OS syntax and add the translated configuration data as a transient change in the checkout configuration during the commit operation. Translation scripts are enabled by default as soon as you add the scripts and related YANG modules to the device using the appropriate operational command.

Use this command to temporarily disable translation scripts for a package to help troubleshoot translation issues instead of deleting the entire package, which would remove the associated data models from the Junos OS schema as well as remove the package and related files from the device. After you disable translation for a package and commit the configuration, the configuration data associated with the YANG data models in that package can be present in the active configuration, but the configuration has no impact on the functioning of the device.

When translation is disabled, you can still configure and commit the statements and hierarchies in the data models added by that package. However, the device does not commit the corresponding Junos OS configuration statements as transient changes during the commit operation for any statements in the data models added by that package, even for those statements that were committed prior to disabling translation.

NOTE: When you disable translation for a package, the device retains any transient configuration changes that were committed prior to disabling translation until the next commit operation.

TIP: Use the show system yang package *package-name* command to verify the translation status of a package.

Options ***package-name***—Name of the YANG package for which to disable translation.

Required Privilege Level maintenance

Related Documentation

- Disabling and Enabling YANG Translation Scripts on Devices Running Junos OS on page 311
- Creating Translation Scripts for YANG Configuration Models on page 308
- request system yang enable on page 443
- show system yang package on page 453

Sample Output

request system yang disable

```
user@host> request system yang disable p1

user@host>
```

request system yang enable

Syntax **request system yang enable *package-name***

Release Information Command introduced in Junos OS Release 16.1R1 on MX Series and T Series routers.
Command introduced in Junos OS Release 17.1R1 on EX Series and QFX Series switches and PTX Series routers.
Command introduced in Junos OS Release 17.3R1 on SRX345, SRX1500, SRX4100, SRX4200, SRX5400, SRX5600, and SRX5800 devices and vSRX instances.
Command introduced in Junos OS Release 18.1R1 on ACX Series routers.

Description Enable the translation scripts associated with the given YANG package.

Translation scripts convert configuration data corresponding to YANG data models into Junos OS syntax and add the translated configuration data as a transient change in the checkout configuration during the commit operation. Translation scripts are enabled by default as soon as you add the scripts and related YANG modules to the device using the appropriate operational command. Use this command to enable translation scripts that were previously disabled using the **request system yang disable** command.

NOTE: When you enable translation for a package, configuration data that is associated with the YANG data models in that package and that is present in the active configuration does not impact the functioning of the device until the next commit operation.

TIP: Use the **show system yang package *package-name*** command to verify the translation status of a package.

Options ***package-name***—Name of the YANG package for which to enable translation.

Required Privilege Level maintenance

Related Documentation

- Disabling and Enabling YANG Translation Scripts on Devices Running Junos OS on page 311
- Creating Translation Scripts for YANG Configuration Models on page 308
- request system yang disable on page 441
- show system yang package on page 453

Sample Output

request system yang enable

```
user@host> request system yang enable p1

user@host>
```

request system yang update

Syntax

```
request system yang update package-name action-script [scripts]
  deviation-module [modules] module [modules] proxy-xml [file-path-names]
  translation-script [scripts]
```

Release Information

Command introduced in Junos OS Release 16.1R1 on MX Series and T Series routers.
Command introduced in Junos OS Release 17.1R1 on EX Series and QFX Series switches and PTX Series routers.
Command introduced in Junos OS Release 17.3R1 on SRX345, SRX1500, SRX4100, SRX4200, SRX5400, SRX5600, and SRX5800 devices and vSRX instances.
proxy-xml option introduced in Junos OS Release 17.3R1 on MX Series and PTX Series routers.
Command introduced in Junos OS Release 18.1R1 on ACX Series routers.

Description

Update an existing YANG package to include new or modified YANG modules or scripts, and merge the updated data models in that package with the Junos OS schema.

When you update a package, the device stores copies of the new and modified module and script files. Junos OS then rebuilds its schema to include the changes to the data models and validates the active configuration against this schema.

NOTE: Devices that use the ephemeral configuration database will delete all ephemeral configuration data in the process of rebuilding the schema.

NOTE: To prevent CLI-related or configuration database errors, we recommend that you do not perform any CLI operations or change the configuration while a device is in the process of adding, updating, or deleting a YANG package and modifying the schema.

NOTE: Starting in Junos OS Release 18.3R1, adding, deleting, or updating YANG packages in configuration mode with the run command is not supported.

Options

package-name—Name of the YANG package to update.

action-script [*scripts*]—List of paths for one or more action scripts to add to or update in the package.

deviation-module [*modules*]—List of paths for one or more deviation modules to add to or update in the package.

module [*modules*]—List of paths for one or more YANG modules to add to or update in the package.

proxy-xml [*file-path-names*]—List of paths for one or more YANG modules to add to or update in the package that provide user-defined OpenConfig mappings for the XML Proxy process to translate Junos Telemetry Interface statistics exported through gRPC into key-value pairs.

translation-script [*scripts*]—List of paths for one or more translation scripts to add to or update in the package.

Required Privilege Level maintenance

Related Documentation

- Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS on page 299
- Configure a Telemetry Sensor in Junos on page 387
- request system yang add on page 436
- show system yang package on page 453

Sample Output

request system yang update

```
user@host> request system yang update p1 module yang/if.yang

YANG modules validation : START
YANG modules validation : SUCCESS
TLV generation: START
TLV generation: SUCCESS
Building schema and reloading /config/juniper.conf.gz ...
Activating /config/juniper.conf.gz ...
mgd: commit complete
Restarting mgd ...

WARNING: cli has been replaced by an updated version:
CLI release 16.1R1 built by builder on 2016-03-30 13:46:11 UTC
Restart cli using the new version ? [yes,no] (yes) yes

Restarting cli ...
```

request system yang validate

Syntax

```
request system yang validate action-script [scripts] module [modules] proxy-xml
  module [modules] translation-script [scripts]
```

Release Information

Command introduced in Junos OS Release 16.1R1 on MX Series and T Series routers.
Command introduced in Junos OS Release 17.1R1 on EX Series and QFX Series switches and PTX Series routers.
Command introduced in Junos OS Release 17.3R1 on SRX345, SRX1500, SRX4100, SRX4200, SRX5400, SRX5600, and SRX5800 devices and vSRX instances.
proxy-xml option introduced in Junos OS Release 17.3R1 on MX Series and PTX Series routers.
Command introduced in Junos OS Release 18.1R1 on ACX Series routers.

Description

Validate the syntax of one or more YANG modules, translation scripts, or action scripts.

Options

action-script *scripts*—List of paths for one or more action scripts to validate.

module *modules*—List of paths for one or more YANG modules to validate.

proxy-xml module *modules*—List of paths for one or more YANG modules to validate that provide user-defined OpenConfig mappings for the XML Proxy process to translate Junos Telemetry Interface statistics exported through gRPC into key-value pairs.

translation-script *scripts*—List of paths for one or more translation scripts to validate.

Required Privilege Level

maintenance

Related Documentation

- Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS on page 299
- Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS on page 297
- Configure a Telemetry Sensor in Junos on page 387

Sample Output

request system yang validate

```
user@host> request system yang validate module [yang/if.yang yang/if-aggregate.yang]
translation-script translation/if.slax
YANG modules validation : START
YANG modules validation : SUCCESS
Scripts syntax validation : START
```

```
script check succeeds
Scripts syntax validation : SUCCESS
```

show system schema

Syntax

```
show system schema module module output-directory path
<filter [filter1 filter2]>
<format format>
<module-name output-module-name>
<output-file-name path>
<version version>
```

Release Information

Command introduced in Junos OS Release 14.2 on EX Series, M Series, MX Series, PTX Series, and T Series.
Command introduced in Junos OS Release 15.1X49-D20 on SRX Series.
Command introduced in Junos OS Release 15.1X53-D30 on QFX Series.
Command introduced in Junos OS Release 15.1X54-D20 on ACX Series.
filter, **module-name**, and **output-directory** options added in Junos OS Release 16.1R1.
output-file-name option deprecated in Junos OS Release 16.1R1.
filter and **module-name** options deprecated in Junos OS Release 17.4R1.

Description

Display the Junos OS schema in the specified format. If you do not specify a format, the device displays the schema in YANG.

Options

filter [*filter1 filter2*]—Display the schema only for the specified configuration hierarchies when generating the schema for the **configuration** module. Specify a space-delimited list of filters representing each hierarchy to include. When you use the **filter** option, you must also include the **module-name** option. This option is deprecated starting in Junos OS Release 17.4R1

NOTE: In the filter path, the root element represents the top-level **configuration** element in the configuration hierarchy. For example, to only retrieve the **[edit system services]** hierarchy, set the value of **filter** to **/system/services**.

format *format*—(Optional) Data modeling language of the schema. Specify **yang** to display the schema in YANG format.

Default: yang

output-directory *path*—Specify the directory where the schema files will be saved.

NOTE: Starting in Junos OS Release 19.2R1, you must specify the **output-directory** option when requesting any schema files. In earlier releases, you can omit the **output-directory** option when requesting a single module to display the module in standard output.

NOTE: To specify the output file in Junos OS Release 15.1 and earlier releases, use the output-file-name option.

output-file-name ***path***—(Optional) File to which the output is written. If you do not specify an absolute path, the device places the file in the current working directory, which defaults to the user's home directory in **/var/home**. If you omit this option, the output is sent to standard output. This option is deprecated starting in Junos OS Release 16.1.

module ***module***—Module for which to display the schema.

module-name ***output-module-name***—(Optional) Name used for the generated module. If you also include the **output-directory** option in the command to direct the output to a file, the filename for the output file uses this module name as the filename base and the format as the file extension. This option is deprecated starting in Junos OS Release 17.4R1.

Required Privilege Level view

Related Documentation

- Understanding YANG on Devices Running Junos OS on page 277
- Understanding Junos OS YANG Modules on page 278
- YANG Modules Overview on page 284
- Using Juniper Networks YANG Modules on page 293

Sample Output

show system schema module all-conf

```
user@host> show system schema module all-conf output-directory /var/tmp/yang

user@host>
```

show system schema module (Junos OS Release 19.1 and earlier)

```
user@host> show system schema module junos-conf-root

/*
 * Copyright (c) 2017 Juniper Networks, Inc.
 * All rights reserved.
 */
 module junos-conf-root {
   namespace "http://yang.juniper.net/junos/conf/root";

   prefix jc;
```

```
  import junos-common-types {
    prefix jt;
  }

  organization "Juniper Networks, Inc.";

  contact "yang-support@juniper.net";

  description "Junos YANG module for configuration hierarchies.";

  revision 2017-01-01 {
    description "Junos: 17.4R1.17";
  }

  container configuration {
    config true;
    uses juniper-config;
    list groups {
      key name;
      ordered-by user;
      description "Configuration groups";
      uses juniper-group;
    }
  }
...
```

show system schema module configuration filter (Junos OS Release 17.4 and earlier)

user@host> **show system schema module configuration filter [/system /interfaces] module-name config-sys-int**

```
/*
 * Copyright (c) 2016 Juniper Networks, Inc.
 * All rights reserved.
 */

 module config-sys-int {
  namespace "http://yang.juniper.net/yang/1.1/jc/configuration/junos/17.2R1.13";

  prefix jc;
  import junos-extension {
    prefix junos;
  }
  ...
   container configuration {
     config true;
     uses juniper-config;
     list groups {
       key group_name;
       ordered-by user;
       description "Configuration groups";
       uses juniper-group;
     }
   }
   grouping juniper-config {
     container system {
       description "System parameters";
       uses juniper-system;
     }
```

```
  container interfaces {
    description "Interface configuration";
    uses apply-advanced;
...
```

show system yang package

Syntax

show system yang package
<package-name>

Release Information

Command introduced in Junos OS Release 16.1R1 on MX Series and T Series routers.
Command introduced in Junos OS Release 17.1R1 on EX Series and QFX Series switches and PTX Series routers.
Command introduced in Junos OS Release 17.3R1 on SRX345, SRX1500, SRX4100, SRX4200, SRX5400, SRX5600, and SRX5800 devices and vSRX instances.
Command introduced in Junos OS Release 18.1R1 on ACX Series routers.

Description

Display YANG packages that are installed on the device and list the corresponding modules, action scripts, and translation scripts associated with the package. The output also includes the translation status for the package, which determines whether the device invokes the package's translation scripts during a commit operation.

Options

none—Display information about all YANG packages that are installed on the device.

package-name—Name of a specific YANG package for which to display information.

Required Privilege Level

view

Related Documentation

- Managing YANG Packages, Modules, and Scripts on Devices Running Junos OS on page 299
- Understanding the Management of Nonnative YANG Modules on Devices Running Junos OS on page 297
- Disabling and Enabling YANG Translation Scripts on Devices Running Junos OS on page 311

Output Fields

Table 15 on page 453 lists the output fields for the **show system yang package** command. Output fields are listed in the approximate order in which they appear.

Table 15: show system yang package Output Fields

Field Name	Field Description
Package ID	Package identifier.
YANG Module(s)	List of YANG modules and deviation modules associated with the package ID. Data models defined in the modules are merged with the Junos OS schema.
Action Script(s)	List of action scripts associated with the package ID.

Table 15: show system yang package Output Fields (continued)

Field Name	Field Description
Translation Script(s)	List of translation scripts associated with the package ID.
Translation script status	Specifies whether translation for that package is **enabled** or **disabled**. When translation is disabled, the device does not invoke the translation scripts for that package during a commit operation, and the changes made to the configuration statements and hierarchies added by that package are not translated into Junos OS syntax and committed as transient configuration changes.

Sample Output

```
user@host> show system yang package p1
Package ID            :p1
YANG Module(s)        :if.yang if-aggregate.yang if-show.yang
Action Script(s)      :if-show.py
Translation Script(s) :if.slax
Translation script status is enabled
```