



Junos[®] OS

NETCONF Java Toolkit Guide

Release
12.3



Published: 2012-11-05

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

This product includes the Envoy SNMP Engine, developed by Epilogue Technology, an Integrated Systems Company. Copyright © 1986-1997, Epilogue Technology Corporation. All rights reserved. This program and its documentation were developed at private expense, and no part of them is in the public domain.

This product includes memory allocation software developed by Mark Moraes, copyright © 1988, 1989, 1993, University of Toronto.

This product includes FreeBSD software developed by the University of California, Berkeley, and its contributors. All of the documentation and software included in the 4.4BSD and 4.4BSD-Lite Releases is copyrighted by the Regents of the University of California. Copyright © 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994. The Regents of the University of California. All rights reserved.

GateD software copyright © 1995, the Regents of the University. All rights reserved. Gate Daemon was originated and developed through release 3.0 by Cornell University and its collaborators. Gated is based on Kirton's EGP, UC Berkeley's routing daemon (routed), and DCN's HELLO routing protocol. Development of Gated has been supported in part by the National Science Foundation. Portions of the GateD software copyright © 1988, Regents of the University of California. All rights reserved. Portions of the GateD software copyright © 1991, D. L. S. Associates.

This product includes software developed by Maker Communications, Inc., copyright © 1996, 1997, Maker Communications, Inc.

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

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

Products made or sold by Juniper Networks or components thereof might be covered by one or more of the following patents that are owned by or licensed to Juniper Networks: U.S. Patent Nos. 5,473,599, 5,905,725, 5,909,440, 6,192,051, 6,333,650, 6,359,479, 6,406,312, 6,429,706, 6,459,579, 6,493,347, 6,538,518, 6,538,899, 6,552,918, 6,567,902, 6,578,186, and 6,590,785.

Junos® OS NETCONF Java Toolkit Guide

12.3

Copyright © 2012, Juniper Networks, Inc.
All rights reserved.

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

YEAR 2000 NOTICE

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

END USER LICENSE AGREEMENT

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

Abbreviated Table of Contents

	About This Guide	ix
Part 1	Overview and Installation	
Chapter 1	NETCONF Java Toolkit Overview	3
Chapter 2	NETCONF Java Toolkit Installation	7
Part 2	Using the NETCONF Java Toolkit	
Chapter 3	NETCONF Java Toolkit Classes	11
Chapter 4	NETCONF Java Toolkit Program Files	21
Chapter 5	Performing Operational and Configuration Tasks	33
Chapter 6	NETCONF Java Toolkit Examples	49
Part 3	Index	

Table of Contents

	About This Guide	ix
	Junos OS Documentation and Release Notes	ix
	Objectives	ix
	Audience	x
	Supported Platforms	x
	Documentation Conventions	x
	Documentation Feedback	xii
	Requesting Technical Support	xii
	Self-Help Online Tools and Resources	xii
	Opening a Case with JTAC	xiii
Part 1	Overview and Installation	
Chapter 1	NETCONF Java Toolkit Overview	3
	NETCONF XML Management Protocol and Junos XML API Overview	3
	NETCONF Java Toolkit Overview	4
Chapter 2	NETCONF Java Toolkit Installation	7
	Downloading and Installing the NETCONF Java Toolkit	7
	Downloading the NETCONF Java Toolkit	7
	Installing the NETCONF Java Toolkit	7
	Satisfying Requirements for SSHv2 Connections	8
Part 2	Using the NETCONF Java Toolkit	
Chapter 3	NETCONF Java Toolkit Classes	11
	NETCONF Java Toolkit Class: Device	11
	NETCONF Java Toolkit Class: NetconfSession	13
	NETCONF Java Toolkit Class: XMLBuilder	14
	NETCONF Java Toolkit Class: XML	15
Chapter 4	NETCONF Java Toolkit Program Files	21
	Creating NETCONF Java Toolkit Program Files	21
	Creating and Executing a NETCONF Java Toolkit Program File	21
	Creating a NETCONF Java Toolkit Program File	22
	Compiling and Executing a NETCONF Java Toolkit Program File	23
	Example: Using the NETCONF Java Toolkit to Execute an Operational Request RPC	24
	Using the NETCONF Java Toolkit to Parse an RPC Reply	27
	Troubleshooting Exception Errors in a NETCONF Java Toolkit Program	30
	Troubleshooting Connection Errors: Socket Timed Out	30
	Troubleshooting Connection Errors: No Connection	30

	Troubleshooting Authentication Errors	31
	Troubleshooting NETCONF Session Errors	31
Chapter 5	Performing Operational and Configuration Tasks	33
	Using the NETCONF Java Toolkit to Perform Operational Tasks	33
	Using Device Object Methods to Execute RPCs and Operational	
	Commands	33
	Executing RPCs	33
	Executing Operational Mode Commands	34
	Example: Using the NETCONF Java Toolkit to Execute CLI Commands	34
	Using the NETCONF Java Toolkit to Perform Configuration Tasks	37
	Using Device Object Methods to Load Configuration Changes	37
	Example: Using the NETCONF Java Toolkit to Load and Commit a	
	Configuration	40
	Example: Using the NETCONF Java Toolkit to Load Set Configuration	
	Commands	43
Chapter 6	NETCONF Java Toolkit Examples	49
	Example: Using the NETCONF Java Toolkit to Execute an Operational Request	
	RPC	49
	Example: Using the NETCONF Java Toolkit to Print Component	
	Temperatures	53
Part 3	Index	
	Index	59

List of Tables

	About This Guide	ix
	Table 1: Notice Icons	x
	Table 2: Text and Syntax Conventions	xi
Part 1	Overview and Installation	
Chapter 1	NETCONF Java Toolkit Overview	3
	Table 3: NETCONF Java Toolkit Classes	4
Part 2	Using the NETCONF Java Toolkit	
Chapter 3	NETCONF Java Toolkit Classes	11
	Table 4: Creating a Configuration Hierarchy with XMLBuilder and XML Objects	16

About This Guide

This preface provides the following guidelines for using the *Junos[®] OS NETCONF Java Toolkit Guide*:

- [Junos OS Documentation and Release Notes on page ix](#)
- [Objectives on page ix](#)
- [Audience on page x](#)
- [Supported Platforms on page x](#)
- [Documentation Conventions on page x](#)
- [Documentation Feedback on page xii](#)
- [Requesting Technical Support on page xii](#)

Junos OS Documentation and Release Notes

For a list of related Junos OS documentation, see <http://www.juniper.net/techpubs/software/junos/>.

If the information in the latest release notes differs from the information in the documentation, follow the *Junos OS Release Notes*.

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

Juniper Networks supports a technical book program to publish books by Juniper Networks engineers and subject matter experts with book publishers around the world. These books go beyond the technical documentation to explore the nuances of network architecture, deployment, and administration using the Junos operating system (Junos OS) and Juniper Networks devices. In addition, the Juniper Networks Technical Library, published in conjunction with O'Reilly Media, explores improving network security, reliability, and availability using Junos OS configuration techniques. All the books are for sale at technical bookstores and book outlets around the world. The current list can be viewed at <http://www.juniper.net/books>.

Objectives

This guide describes how to use the NETCONF Java toolkit to establish a NETCONF session with and to configure or request information from a remote routing, switching, or security device.

This guide is an introduction to the NETCONF Java toolkit and provides a limited overview of the methods available in the toolkit. For information about the methods and full functionality available in the toolkit, consult the NETCONF Java toolkit Javadocs.



NOTE: For additional information about the Junos OS—either corrections to or information that might have been omitted from this guide—see the software release notes at <http://www.juniper.net/>.

Audience

This guide is designed for software developers who want to write custom Java programs that use NETCONF to remotely operate, configure, and monitor Juniper Networks devices running Junos OS. This guide assumes that you are familiar with the Java programming language and object-oriented programming concepts. It also assumes that you are familiar with basic terminology and concepts of the Extensible Markup Language (XML), with XML-parsing utilities such as the Document Object Model (DOM) or Simple API for XML (SAX), and with the Junos OS command-line interface (CLI). For more information about XML and Junos OS, see [Day One: Navigating the Junos XML Hierarchy](#).

To use this guide, you need a broad understanding of networks in general, the Internet in particular, networking principles, and network configuration.

Supported Platforms

The NETCONF Java toolkit can be used to request and change configuration information on routing, switching, and security platforms running Junos OS.

Documentation Conventions

Table 1 on page x 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.

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

Table 2: Text and Syntax Conventions

Convention	Description	Examples
Bold text like this	Represents text that you type.	To enter configuration mode, type the configure command: user@host> configure
Fixed-width text like this	Represents output that appears on the terminal screen.	user@host> show chassis alarms No alarms currently active
<i>Italic text like this</i>	<ul style="list-style-type: none"> Introduces or emphasizes important new terms. Identifies book names. Identifies RFC and Internet draft titles. 	<ul style="list-style-type: none"> A policy <i>term</i> is a named structure that defines match conditions and actions. <i>Junos OS System Basics Configuration Guide</i> RFC 1997, <i>BGP Communities Attribute</i>
<i>Italic text like this</i>	Represents variables (options for which you substitute a value) in commands or configuration statements.	Configure the machine's domain name: [edit] root@# set system domain-name <i>domain-name</i>
Text like this	Represents names of configuration statements, commands, files, and directories; configuration hierarchy levels; or labels on routing platform components.	<ul style="list-style-type: none"> To configure a stub area, include the stub statement at the [edit protocols ospf area area-id] hierarchy level. The console port is labeled CONSOLE.
< > (angle brackets)	Enclose optional keywords or variables.	stub <default-metric <i>metric</i> >;
(pipe symbol)	Indicates a choice between the mutually exclusive keywords or variables on either side of the symbol. The set of choices is often enclosed in parentheses for clarity.	broadcast multicast (<i>string1</i> <i>string2</i> <i>string3</i>)
# (pound sign)	Indicates a comment specified on the same line as the configuration statement to which it applies.	rsvp { # Required for dynamic MPLS only
[] (square brackets)	Enclose a variable for which you can substitute one or more values.	community name members [<i>community-ids</i>]
Indentation and braces ({ })	Identify a level in the configuration hierarchy.	[edit] routing-options { static { route default { nexthop <i>address</i> ; retain; } } }
;(semicolon)	Identifies a leaf statement at a configuration hierarchy level.	

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

Convention	Description	Examples
J-Web GUI Conventions		
Bold text like this	Represents J-Web graphical user interface (GUI) items you click or select.	<ul style="list-style-type: none"> In the Logical Interfaces box, select All Interfaces. To cancel the configuration, click Cancel.
> (bold right angle bracket)	Separates levels in a hierarchy of J-Web selections.	In the configuration editor hierarchy, select Protocols>Ospf .

Documentation Feedback

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

- Document or topic name
- URL or page number
- Software release version (if applicable)

Requesting Technical Support

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

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

Self-Help Online Tools and Resources

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

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

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

Opening a Case with JTAC

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

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

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

PART 1

Overview and Installation

- [NETCONF Java Toolkit Overview on page 3](#)
- [NETCONF Java Toolkit Installation on page 7](#)

CHAPTER 1

NETCONF Java Toolkit Overview

- NETCONF XML Management Protocol and Junos XML API Overview on page 3
- NETCONF Java Toolkit Overview on page 4

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. The NETCONF XML management protocol 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 CLI, which administrators use to retrieve status information for a routing platform running Junos OS.

The NETCONF XML management protocol is described in RFC 4741, *NETCONF Configuration Protocol*, which is available at <http://www.ietf.org/rfc/rfc4741.txt>.

Client applications request or change information 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 **<get-interface-information>** tag element from the Junos XML API. The NETCONF server gathers the information from the interface process and returns it in the **<interface-information>** 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, but 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.

NETCONF Java Toolkit Overview

The NETCONF Java toolkit provides an object-oriented interface for communicating with a NETCONF server. The toolkit enables programmers familiar with the Java programming language to easily connect to a device, open a NETCONF session, construct configuration hierarchies in XML, and create and execute operational and configuration requests. You can create your own Java applications to manage and configure routing, switching, and security devices.

The NETCONF Java toolkit provides classes with methods that implement the functionality of the NETCONF protocol operations defined in [RFC 4741](#). All basic protocol operations are supported. The NETCONF XML management protocol uses XML-based data encoding for configuration data and remote procedure calls. The toolkit provides classes and methods that aid in creating, modifying, and parsing XML.

The NETCONF Java toolkit has four basic classes, which are described in [Table 3 on page 4](#).

Table 3: NETCONF Java Toolkit Classes

Class	Summary
Device	Defines the device on which the NETCONF server runs, and represents the SSHv2 connection and default NETCONF session with that device.
NetconfSession	Represents a NETCONF session established with the device on which the NETCONF server runs.
XMLBuilder	Creates XML-encoded data.
XML	XML-encoded data that represents an operational or configuration request or configuration data.

A *configuration management server* is generally a PC or workstation that is used to configure a router, switch, or security device remotely. The communication between the configuration management server and the NETCONF server via the NETCONF Java toolkit involves:

- Establishing a NETCONF session over SSHv2 between the configuration management server and the NETCONF server.
- Creating RPCs corresponding to requests and sending these requests to the NETCONF server.
- Receiving and processing the RPC replies from the NETCONF server.

To use the NETCONF Java toolkit, you must install the toolkit and add the `.jar` path to your CLASSPATH. For more information about installing the NETCONF Java toolkit, see [“Downloading and Installing the NETCONF Java Toolkit” on page 7](#).

Once the toolkit is installed, you connect to a device, create a NETCONF session, and execute operations by adding the associated code to a Java program file, which is then compiled and executed. For more information about creating NETCONF Java toolkit programs, see [“Creating and Executing a NETCONF Java Toolkit Program File” on page 21](#).



NOTE: Juniper Networks devices running Junos OS Release 7.5R1 or later support the NETCONF XML management protocol.

**Related
Documentation**

- [NETCONF Java Toolkit Class: Device on page 11](#)
- [NETCONF Java Toolkit Class: NetconfSession on page 13](#)
- [NETCONF Java Toolkit Class: XML on page 15](#)
- [NETCONF Java Toolkit Class: XMLBuilder on page 14](#)
- [NETCONF XML Management Protocol and Junos XML API Overview on page 3](#)
- [Downloading and Installing the NETCONF Java Toolkit on page 7](#)
- [Creating and Executing a NETCONF Java Toolkit Program File on page 21](#)

CHAPTER 2

NETCONF Java Toolkit Installation

- [Downloading and Installing the NETCONF Java Toolkit on page 7](#)

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 7](#)
2. [Installing the NETCONF Java Toolkit on page 7](#)
3. [Satisfying Requirements for SSHv2 Connections on page 8](#)

Downloading the NETCONF Java Toolkit

To download the NETCONF Java toolkit to the configuration management server:

1. Access the download page at <http://www.juniper.net/support/downloads/?p=netconf>.
2. Select the appropriate software release.
3. Select the Software tab.
4. Click the NETCONF API Java Toolkit link, and download the file to the configuration management server.

Installing the NETCONF Java Toolkit

To install the NETCONF Java toolkit on the configuration management server:

1. Unzip the NETCONF API Java Toolkit zip file downloaded in the previous section.
2. Include the **Netconf.jar** file in the CLASSPATH of your local Java development environment.

For more information about setting the CLASSPATH, see <http://download.oracle.com/javase/1.4.2/docs/tooldocs/windows/classpath.html>.

3. 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 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 Toolkit Program File on page 21](#)
- [NETCONF Java Toolkit Overview on page 4](#)
- [NETCONF XML Management Protocol and Junos XML API Overview on page 3](#)

PART 2

Using the NETCONF Java Toolkit

- [NETCONF Java Toolkit Classes on page 11](#)
- [NETCONF Java Toolkit Program Files on page 21](#)
- [Performing Operational and Configuration Tasks on page 33](#)
- [NETCONF Java Toolkit Examples on page 49](#)

CHAPTER 3

NETCONF Java Toolkit Classes

- [NETCONF Java Toolkit Class: Device on page 11](#)
- [NETCONF Java Toolkit Class: NetconfSession on page 13](#)
- [NETCONF Java Toolkit Class: XMLBuilder on page 14](#)
- [NETCONF Java Toolkit Class: XML on page 15](#)

NETCONF Java Toolkit Class: Device

A **net.juniper.netconf.Device** object represents an SSHv2 connection and a default NETCONF session between the configuration management server and the device on which the NETCONF server resides.

When creating a **Device** object, you must provide the IP address or DNS name and the authentication details to create the SSHv2 connection. Authentication can be user-password based or RSA/DSA key-based. You also have the option of specifying the port number for the SSHv2 connection and the client capabilities to be sent to the NETCONF server.

The constructor syntax is:

```
Device (String hostname, String login, String password, String pemKeyFile)  
Device (String hostname, String login, String password, String pemKeyFile, int port)  
Device (String hostname, String login, String password, String pemKeyFile,  
        ArrayList capabilities)  
Device (String hostname, String login, String password, String pemKeyFile, int port,  
        ArrayList capabilities)
```

The constructor parameters are:

- **hostname**—(Required) IP address or DNS name of the device on which the NETCONF server is running and to which to connect via SSHv2.
- **login**—(Required) Username for the login account on the device on which the NETCONF server is running.
- **password**—(Required) Password for either user password-based authentication or key-based authentication. If no password is required for key-based authentication, pass this argument as null.

- **pemKeyFile**—(Required) Path of the file containing the DSA/RSA private key in PEM format for key-based authentication. For user password-based authentication, pass this argument as null.
- **port**—(Optional) Port number on which to establish the SSHv2 connection. The default port is 830. If you are connecting to a device that is configured for NETCONF over SSH on a port other than the default port, you must specify that port number in the arguments.
- **capabilities**—(Optional) Client capabilities to be communicated to the NETCONF server, if the capabilities are other than the default capabilities.

The default capabilities sent to the NETCONF server are:

```
urn:ietf:params:xml:ns:netconf:base:1.0
urn:ietf:params:xml:ns:netconf:base:1.0#candidate
urn:ietf:params:xml:ns:netconf:base:1.0#confirmed-commit
urn:ietf:params:xml:ns:netconf:base:1.0#validate
urn:ietf:params:xml:ns:netconf:base:1.0#url?protocol=http,ftp,file
```

The general syntax for creating a **Device** object is:

```
Device device_name = new Device (String hostname, String login,
                                String password, String pemKeyFile,
                                <int port>, <ArrayList capabilities>)
```

By default, a **NetconfSession** object is created when you create a new instance of **Device** and connect to a NETCONF server. Once you have created a **Device** object, you can perform NETCONF operations.

Examples The following example creates a **Device** object with an authenticated SSHv2 connection to IP address 10.10.1.1. The connection uses user password-based authentication with the login name "admin" and the password "PaSsWoRd". When the **connect()** method is called, it connects to the device and automatically establishes a default NETCONF session.

```
Device my_device = new Device("10.10.1.1", "admin", "PaSsWoRd", null);
my_device.connect();
```

To create a **Device** object with a NETCONF-over-SSH connection on port 49000 instead of the default port 830, add the port number to the constructor arguments.

```
Device my_device = new Device("10.10.1.1", "admin", "PaSsWoRd", null, 49000);
```

The default timeout value for connecting to the device is 5000 milliseconds. To set the timeout value to a different interval, call the **setTimeout()** method on the device object.

**Related
Documentation**

- [Using Device Object Methods to Execute RPCs and Operational Commands on page 33](#)
- [Creating and Executing a NETCONF Java Toolkit Program File on page 21](#)
- [Troubleshooting Exception Errors in a NETCONF Java Toolkit Program on page 30](#)
- [NETCONF Java Toolkit Class: NetconfSession on page 13](#)
- [NETCONF Java Toolkit Class: XML on page 15](#)
- [NETCONF Java Toolkit Class: XMLBuilder on page 14](#)

- [NETCONF Java Toolkit Overview on page 4](#)

NETCONF Java Toolkit Class: `NetconfSession`

A `net.juniper.netconf.NetconfSession` object represents the NETCONF session between the configuration management server and the device on which the NETCONF server resides.

By default, a NETCONF session is created when you create a new instance of `Device` and connect to a NETCONF server, so you do not need to explicitly create a `NetconfSession` object. You can perform the NETCONF operations directly from the `Device` object by calling the associated methods.

However, there might be times when you need multiple NETCONF sessions on the same SSHv2 connection. To create multiple sessions, call the `createNetconfSession()` method on the `Device` object as shown in the following example:

```
Device device = new Device("10.10.1.1", "admin", "PaSsWoRd", null);
device.connect();
NetconfSession second_session = device.createNetconfSession();
```

Once you create an additional NETCONF session, you call the NETCONF operation methods for the new `NetconfSession` object in the same way as you call them for the `Device` object.

The `Device` and `NetconfSession` classes contain many identical methods, which perform NETCONF operations such as executing remote procedure calls (RPCs) and performing configuration changes. When you call a method on the `Device` object, it acts on the default NETCONF session. When you call a method on any additional `NetconfSession` object, it acts on that NETCONF session.

Example: Creating Multiple NETCONF Sessions

In the following example, the code snippet creates a new `Device` object. When the `connect()` method is called, the program connects to the remote device and establishes a default NETCONF session. The program creates a second `NetconfSession` object, `second_session`. Calling `device.getSessionID()` returns the session ID of the default NETCONF session, and calling `second_session.getSessionID()` returns the session ID of the second NETCONF session.

```
// Create a device object and a default NETCONF session
Device device = new Device("10.10.1.34", "admin", "PaSsWoRd", null);
device.connect();

// Create an additional NETCONF session
NetconfSession second_session = device.createNetconfSession();

// There are two independent NETCONF sessions
String default_session_id = device.getSessionID();
String second_session_id = second_session.getSessionID();
```

Related Documentation

- [NETCONF Java Toolkit Class: `Device` on page 11](#)
- [NETCONF Java Toolkit Class: `XML` on page 15](#)

- [NETCONF Java Toolkit Class: XMLBuilder on page 14](#)
- [NETCONF Java Toolkit Overview on page 4](#)

NETCONF Java Toolkit Class: XMLBuilder

In a NETCONF session, communication between the configuration management server and the NETCONF server is through XML-encoded data. The configuration management server sends remote procedure calls (RPCs) to the NETCONF server, and the NETCONF server processes the RPC and returns an RPC reply. The `net.juniper.netconf.XMLBuilder` and `net.juniper.netconf.XML` objects help create and parse XML-encoded data.

You use the XMLBuilder object to create a new XML object. The constructor syntax is:

```
XMLBuilder ()
```

The XMLBuilder class includes methods to create a configuration hierarchy, an RPC, or an XML object as XML-encoded data. Each method is overloaded to accept multiple hierarchy levels. The methods return an XML object. For example, the methods to construct a configuration, RPC, or XML object with a single-tier hierarchy are:

- `createNewConfig(String elementLevelOne)`
- `createNewRPC(String elementLevelOne)`
- `createNewXML(String elementLevelOne)`

The following sample code creates a new `XMLBuilder` object, `builder`. The `XMLBuilder` object calls the `createNewConfig()` method to construct a three-tier configuration hierarchy consisting of a “security” element, a “policies” element child tag, and a “policy” element that is a child of “policies”.

```
XMLBuilder builder = new XMLBuilder();  
XML policy = builder.createNewConfig("security","policies","policy");
```

The resulting XML hierarchy is as follows.

```
<configuration>  
  <security>  
    <policies>  
      <policy>  
    </policy>  
    </policies>  
  </security>  
</configuration>
```

Notice that the `createNewConfig()` method always encloses the hierarchy within a top-level root element `<configuration>`. Similarly, the `createNewRPC()` method encloses the hierarchy within an `<rpc>` tag element.

Once you generate an XML object, you can call methods from the `XML` class to manipulate that object.

Related Documentation

- [NETCONF Java Toolkit Class: Device on page 11](#)

- [NETCONF Java Toolkit Class: NetconfSession on page 13](#)
- [NETCONF Java Toolkit Class: XML on page 15](#)
- [NETCONF Java Toolkit Overview on page 4](#)

NETCONF Java Toolkit Class: XML

A `net.juniper.netconf.XML` object represents XML-encoded data and provides methods to modify and parse the XML. The XML object internally maintains an `org.w3c.dom.Document` object, corresponding to the XML data it represents.

It is recommended that you work with the XML object to create new configurations, remote procedure calls (RPCs), or any XML-based data. Using an XML object, you can easily add, delete, or modify elements and attributes. To facilitate modification of XML content, the XML object maintains an 'active' element, which represents the hierarchy level exposed for modification.

To create an XML object, you first create an `XMLBuilder` object and construct the initial XML hierarchy. The `XMLBuilder` methods return an XML object on which you can then build. This makes it convenient to create XML-based configurations and RPCs and also parse the XML-based replies received from the NETCONF server.

Example: Creating a Configuration Hierarchy

This example creates the following sample XML configuration hierarchy. The steps used to create the configuration hierarchy are outlined in [Table 4 on page 16](#).

```
<configuration>
  <security>
    <policies>
      <policy>
        <from-zone-name>trust</from-zone-name>
        <to-zone-name>untrust</to-zone-name>
        <policy>
          <name>my-sec-policy</name>
          <match>
            <source-address>any</source-address>
            <destination-address>any</destinationaddress>
            <application>junos-ftp</application>
            <application>junos-ntp</application>
            <application>junos-ssh</application>
          </match>
          <then>
            <permit>
            </permit>
          </then>
        </policy>
      </policy>
    </policies>
  </security>
</configuration>
```

Table 4: Creating a Configuration Hierarchy with XMLBuilder and XML Objects

Java Code	Resulting Hierarchy
<pre>// Create an XMLBuilder object and a 3-level hierarchy XMLBuilder builder = new XMLBuilder(); XML policy = builder.createNewConfig("security","policies","policy");</pre>	<pre><configuration> <security> <policies> <policy> </policy> </policies> </security> </configuration></pre>
<pre>// Append nodes at the 'policy' level policy.append("from-zone-name","trust"); policy.append("to-zone-name","untrust");</pre>	<pre><configuration> <security> <policies> <policy> <from-zone-name>trust</from-zone-name> <to-zone-name>untrust</to-zone-name> </policy> </policies> </security> </configuration></pre>
<pre>// Create a new hierarchy level for the first policy XML policyOne = policy.append("policy"); policyOne.append("name","my-sec-policy");</pre>	<pre><configuration> <security> <policies> <policy> <from-zone-name>trust</from-zone-name> <to-zone-name>untrust</to-zone-name> <policy> <name>my-sec-policy</name> </policy> </policy> </policies> </security> </configuration></pre>

Table 4: Creating a Configuration Hierarchy with XMLBuilder and XML Objects (*continued*)

Java Code	Resulting Hierarchy
<pre>// Create the 'match' hierarchy XML match = policyOne.append("match"); // Create and append an applications array // to make three nodes with the same node name String[] applications = {"junos-ftp","junos-ntp","junos-ssh"}; match.append("application", applications);</pre>	<pre><configuration> <security> <policies> <policy> <from-zone-name>trust</from-zone-name> <to-zone-name>untrust</to-zone-name> <policy> <name>my-sec-policy</name> <match> <application>junos-ftp</application> <application>junos-ntp</application> <application>junos-ssh</application> </match> </policy> </policy> </policies> </security> </configuration></pre>
<pre>// Add elements under 'match' match.append("source-address","any"); match.append("destination-address","any");</pre>	<pre><configuration> <security> <policies> <policy> <from-zone-name>trust</from-zone-name> <to-zone-name>untrust</to-zone-name> <policy> <name>my-sec-policy</name> <match> <application>junos-ftp</application> <application>junos-ntp</application> <application>junos-ssh</application> <source-address>any</source-address> <destination-address>any</destination-address> </match> </policy> </policy> </policies> </security> </configuration></pre>

Table 4: Creating a Configuration Hierarchy with XMLBuilder and XML Objects (*continued*)

Java Code	Resulting Hierarchy
<pre>// Add the 'then' hierarchy with a child 'permit' element policyOne.append("then").append("permit");</pre>	<pre><configuration> <security> <policies> <policy> <from-zone-name>trust</from-zone-name> <to-zone-name>untrust</to-zone-name> <policy> <name>my-sec-policy</name> <match> <application>junos-ftp</application> <application>junos-ntp</application> <application>junos-ssh</application> <source-address>any</source-address> <destination-address>any</destination-address> </match> <then> <permit/> </then> </policy> </policies> </security> </configuration></pre>
<pre>// Complete code and final configuration XMLBuilder builder = new XMLBuilder(); XML policy = builder.createNewConfig("security","policies","policy"); policy.append("from-zone-name","trust"); policy.append("to-zone-name","untrust"); XML policyOne = policy.append("policy"); policyOne.append("name","my-sec-policy"); XML match = policyOne.append("match"); String[] applications = {"junos-ftp","junos-ntp","junos-ssh"}; match.append("application", applications); match.append("source-address","any"); match.append("destination-address","any"); policyOne.append("then").append("permit");</pre>	<pre><configuration> <security> <policies> <policy> <from-zone-name>trust</from-zone-name> <to-zone-name>untrust</to-zone-name> <policy> <name>my-sec-policy</name> <match> <application>junos-ftp</application> <application>junos-ntp</application> <application>junos-ssh</application> <source-address>any</source-address> <destination-address>any</destination-address> </match> <then> <permit/> </then> </policy> </policies> </security> </configuration></pre>

**Related
Documentation**

- [NETCONF Java Toolkit Class: Device on page 11](#)
- [NETCONF Java Toolkit Class: NetconfSession on page 13](#)
- [NETCONF Java Toolkit Class: XMLBuilder on page 14](#)

- [NETCONF Java Toolkit Overview on page 4](#)

CHAPTER 4

NETCONF Java Toolkit Program Files

- [Creating NETCONF Java Toolkit Program Files on page 21](#)
- [Using the NETCONF Java Toolkit to Parse an RPC Reply on page 27](#)
- [Troubleshooting Exception Errors in a NETCONF Java Toolkit Program on page 30](#)

Creating NETCONF Java Toolkit Program Files

- [Creating and Executing a NETCONF Java Toolkit Program File on page 21](#)
- [Example: Using the NETCONF Java Toolkit to Execute an Operational Request RPC on page 24](#)

Creating and Executing a NETCONF Java Toolkit Program File

You can use the NETCONF Java toolkit to connect to a device, open a NETCONF session, and create and execute operational and configuration requests. After installing the NETCONF Java toolkit, which is described in [“Downloading and Installing the NETCONF Java Toolkit” on page 7](#), the general procedure is:

1. Create a Java program that includes code to connect to a device and to execute the desired operations or requests.
2. Compile the Java code and execute the program.

These steps are reviewed in detail in the following sections:

- [Creating a NETCONF Java Toolkit Program File on page 22](#)
- [Compiling and Executing a NETCONF Java Toolkit Program File on page 23](#)

Creating a NETCONF Java Toolkit Program File

NETCONF Java toolkit programs have the same generic framework. To create a basic NETCONF Java toolkit program:

1. Create a **.java** file.

The filename should be identical to the class name, excluding the extension. For example, the **ShowChassis** class is saved in the file **ShowChassis.java**.

2. Create the general boilerplate, which includes the code for import statements, the class declaration, and the Java method, **main()**.

```
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import net.juniper.netconf.Device;
import net.juniper.netconf.NetconfException;
import net.juniper.netconf.XML;
import org.xml.sax.SAXException;

public class ShowChassis {
    public static void main(String args[]) throws NetconfException,
        ParserConfigurationException, SAXException, IOException {

    }
}
```

3. Within **main()**, create a **Device** object and call the **connect()** method.

This also creates a default NETCONF session with the NETCONF server over SSHv2.

```
Device device = new Device("hostname", "username", "password", null);
device.connect();
```

4. Execute operational and configuration requests by executing RPCs and performing NETCONF operations on the **Device** object.

For example, to execute an operational request to retrieve chassis inventory information from the device, include the following line of code:

```
XML reply = device.executeRPC("get-chassis-inventory");
```

5. Add code to print, parse, or take action on RPC replies received from the NETCONF server.

The following line of code prints the RPC reply in XML format to standard output:

```
System.out.println(reply.toString());
```

6. Close the device and release resources by calling the **close()** method on the **Device** object.

```
device.close();
```

Sample NETCONF Java Toolkit Program

The following sample code illustrates a simple NETCONF Java toolkit program, **ShowChassis.java**, which connects to a device and executes an operational request for chassis inventory information:

```
/* ShowChassis.java */
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import net.juniper.netconf.Device;
import net.juniper.netconf.NetconfException;
import net.juniper.netconf.XML;
import org.xml.sax.SAXException;

public class ShowChassis {
    public static void main(String args[]) throws NetconfException,
        ParserConfigurationException, SAXException, IOException {

        //Create the device object and establish a NETCONF session
        Device device = new Device("hostname", "username", "password", null);
        device.connect();

        //Send RPC and receive RPC reply as XML
        XML rpc_reply = device.executeRPC("get-chassis-inventory");

        //Print the RPC reply and close the device
        System.out.println(rpc_reply.toString());
        device.close();
    }
}
```

Compiling and Executing a NETCONF Java Toolkit Program File

To execute a NETCONF Java toolkit program, compile the code and run the program from the configuration management server. You need a Java compiler to compile the source code and to create an executable program.

1. Compile the Java source code to create a Java class file containing Java bytecode.

For example, to compile the **ShowChassis.java** file using the **javac** compiler included in the Java Development Kit (JDK) from Oracle Corporation, you would issue the following command on the command line of the configuration management server:

```
>javac ShowChassis.java
```

This creates the **ShowChassis.class** file.

2. Execute the resulting program.

```
>java ShowChassis
```

Example: Using the NETCONF Java Toolkit to Execute an Operational Request RPC

This NETCONF Java toolkit program executes an RPC to obtain operational information from a device, which is then printed to standard output. This example serves as an instructional example for creating and executing a basic NETCONF Java toolkit program.

- [Requirements on page 24](#)
- [Overview on page 24](#)
- [Configuration on page 24](#)
- [Verification on page 26](#)
- [Troubleshooting on page 26](#)

Requirements

- NETCONF Java toolkit is installed on the configuration management server.
- Client application can log in to the device where the NETCONF server resides.
- NETCONF service over SSH is enabled on the device where the NETCONF server resides.

Overview

You can use the NETCONF Java toolkit to request operational information from a remote device. The following example illustrates how to create a NETCONF Java toolkit program to execute an operational request from the Junos XML API on a device running Junos OS. The example also explains how to compile the code, execute the program, and verify the results.

Configuration

Creating the Java Program

Step-by-Step Procedure

To construct the Java program file that contains the code for the operational request:

1. Give the file a descriptive name.

The filename must be the same as the class name. For this example, the file and class are named **GetChassisInventory**.
2. Include the appropriate import statements, and the code for the class declaration and the Java method, **main()**.

```
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import net.juniper.netconf.Device;
import net.juniper.netconf.NetconfException;
import net.juniper.netconf.XML;
import org.xml.sax.SAXException;

public class GetChassisInventory {
    public static void main(String args[]) throws NetconfException,
        ParserConfigurationException, SAXException, IOException {
    }
}
```

3. Within **main()**, create a **Device** object and call the **connect()** method.

This creates a default NETCONF session over SSHv2 with the NETCONF server. You must update the code with the appropriate arguments for connection to and authentication on your specific device.

```
Device device = new Device("10.10.1.1", "admin", "PaSsWoRd", null);
device.connect();
```

Having established a **Device** object, you can perform NETCONF operations on the device. For a complete list of available methods corresponding to NETCONF operations, refer to the NETCONF Java toolkit Javadocs.

4. Call the **executeRPC()** method with the operational request RPC command as the argument.

This example uses the Junos XML API **get-chassis-inventory** RPC. The reply, which is returned in XML, is stored in the **rpc_reply** variable.

```
XML rpc_reply = device.executeRPC("get-chassis-inventory");
```

5. Add code to take action on the RPC reply.

The following code converts the NETCONF server's reply to a string and prints it to the screen:

```
System.out.println(rpc_reply.toString());
```

6. Close the device and release resources by calling the **close()** method on the device object.

```
device.close();
```

Results The complete program is:

```
/*GetChassisInventory*/
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import net.juniper.netconf.Device;
import net.juniper.netconf.NetconfException;
import net.juniper.netconf.XML;
import org.xml.sax.SAXException;

public class GetChassisInventory {
    public static void main(String args[]) throws NetconfException,
        ParserConfigurationException, SAXException, IOException {

        Device device = new Device("10.10.1.1","admin","PaSsWoRd",null);
        device.connect();
        XML rpc_reply = device.executeRPC("get-chassis-inventory");
        System.out.println(rpc_reply.toString());
        device.close();
    }
}
```

Compiling and Running the Java Program

Step-by-Step Procedure You need a Java compiler to compile the source code and to create an executable program.

To compile the code and run the program on the configuration management server:

1. Compile the **GetChassisInventory.java** file.

```
>javac GetChassisInventory.java
```
2. Execute the resulting **GetChassisInventory** program.

```
>java GetChassisInventory
```

Verification

Verifying Program Execution

Purpose Verify that the GetChassisInventory program runs correctly.

Action If the program executes successfully, it establishes a connection and creates a NETCONF session with the specified device. The program sends the **get-chassis-inventory** RPC to the NETCONF server, and the server responds with the requested operational information enclosed in the **<rpc-reply>** tag element. The program prints the reply to standard out. Following is a sample RPC reply with some output omitted for brevity.

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
  xmlns:junos="http://xml.juniper.net/junos/11.2R1/junos">
  <chassis-inventory xmlns="http://xml.juniper.net/junos/11.2R1/junos-chassis">
  <chassis junos:style="inventory">
  <name>Chassis</name>
  <serial-number>12345</serial-number>
  <description>M7i</description>
  <chassis-module>

...output omitted...

</chassis>
</chassis-inventory>
</rpc-reply>
```

Troubleshooting

- [Troubleshooting NETCONF Exceptions on page 26](#)

Troubleshooting NETCONF Exceptions

Problem A NETCONF exception occurs, and you see the following error message:

```
Exception in thread "main" net.juniper.netconf.NetconfException: There was a
problem while connecting to 10.10.1.1:830
    at net.juniper.netconf.Device.createNetconfSession(Device.java:344)
    at net.juniper.netconf.Device.connect(Device.java:225)
    at GetChassisInventory.main(GetChassisInventory.java:14)
```


NETCONF over SSH might not be enabled on the device where the NETCONF server resides, or it might be enabled on a different port.

Solution Ensure that you have enabled NETCONF over SSH on the device where the NETCONF server resides. Since the example program does not specify a specific port number in the **Device** arguments, the NETCONF session is established on the default NETCONF-over-SSH port, 830. To verify whether NETCONF over SSH is enabled on the default port for a device running Junos OS, enter the following operational mode command on the remote device:

```
user@host> show configuration system services
```

```
ftp;
netconf {
    ssh;
}
```

If the **netconf** configuration hierarchy is absent, issue the following statements in configuration mode to enable NETCONF over SSH on the default port:

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

If the **netconf** configuration hierarchy specifies a port other than the default port, include the new port number in the **Device** object constructor arguments. For example, the following device is configured for NETCONF over SSH on port 12345:

```
user@host> show configuration system services
netconf {
    ssh {
        port 12345;
    }
}
```

To correct the connection issue, include the new port number in the **Device** arguments.

```
Device device = new Device("10.10.1.1", "admin", "PaSsWoRd", null, 12345);
```

Using the NETCONF Java Toolkit to Parse an RPC Reply

After submitting an operational or configuration request to the NETCONF server, the server responds with an RPC reply.

```
XML rpc_reply = device.executeRPC("get-chassis-inventory");
```

There are two approaches to parse an XML reply within the context of the NETCONF Java toolkit:

- Get the **org.w3c.dom.Document** object and use the native parsing methods available in the standard Java class libraries for a **Document** object.
- Use the **findValue(List list)** and **findNodes(List list)** methods available in the **net.juniper.netconf.XML** class on the XML object.

For the first approach, call the `getOwnerDocument()` method on the reply object to return the **Document** object.

```
Document doc = rpc_reply.getOwnerDocument();
```

You can then use methods in the standard Java libraries on the resulting **Document** object. This method is useful for the flexibility and options available in terms of the standard Java library methods.

For the second approach, the `net.juniper.netconf.XML` class contains the `findValue(List list)` and `findNodes(List list)` methods, which you can use to parse the XML object. You must include the "import java.util.*;" statement in your program code to use the functionality of the **List** interface or to create an **Arrays** object as shown in the corresponding examples.

Study the following RPC reply for the `get-interface-information` operational request:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
  xmlns:junos="http://xml.juniper.net/junos/11.3I0/junos">
  <interface-information>
    <physical-interface>
      <name>ge-0/0/0</name>
      <admin-status>up</admin-status>
      <oper-status>up</oper-status>
      /* hierarchy truncated for brevity */
    </physical-interface>
    <physical-interface>
      <name>ge-0/0/1</name>
      /* hierarchy truncated for brevity */
    </physical-interface>
  </interface-information>
</rpc-reply>
```

Parsing an RPC Reply Using `findValue()`

You can use the `findValue()` method to determine the value of a given element at any level of the hierarchy. In the example RPC reply for `get-interface-information`, suppose you want to determine the value of the `<admin-status>` element of the physical interface `ge-0/0/0`. Being aware of the format of the RPC reply, you can extract this information using the following code:

```
XML rpc_reply = device.executeRPC("get-interface-information");
List<String> list = Arrays.asList("interface-information", "physical-interface",
  "name~ge-0/0/0", "admin-status");
String admin_status = rpc_reply.findValue(list);
System.out.println(admin_status);
```

Note that the interface name uses a tilde (~) character to identify the particular element. Execution of this code prints "up" to standard output.

Parsing an RPC Reply Using `findNodes()`

You can use the `findNodes()` method to obtain the list of all nodes under a given hierarchy as `org.w3c.dom.Node` objects. The following code snippet obtains a list of all `<physical-interface>` nodes under the `<interface-information>` element in the hierarchy:

```
XML rpc_reply = device.executeRPC("get-interface-information");
List<String> list = Arrays.asList("interface-information", "physical-interface");
List physical_interfaces_list = rpc_reply.findNodes(list);
```

However, you might want to extract a specific node. The following code returns the hierarchy for the ge-0/0/1 interface only:

```
XML rpc_reply = device.executeRPC("get-interface-information");
List<String> list = Arrays.asList("interface-information", "physical-interface",
    "name~ge-0/0/1");
List physical_interfaces_list = rpc_reply.findNodes(list);
Node ge001_node = (Node)physical_interfaces_list.get(0);
```

**Example: Parsing an
RPC Reply Using
findNodes() (Detailed)**

The following example takes this approach a step further and parses through the child nodes to extract and print the content for just the `<name>` elements. This sample code focuses on the portion of the program that parses the RPC reply and does not represent a complete program.

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

/* code omitted for brevity */

XML rpc_reply = device.executeRPC("get-interface-information");

// Obtain a list of list of 'org.w3c.dom.Node' objects
List<String> list = Arrays.asList("interface-information", "physical-interface");
List physical_interfaces_list = rpc_reply.findNodes(list);

// Print the value for each of the name elements:
Iterator iter = physical_interfaces_list.iterator();
while(iter.hasNext()) {
    Node node = (Node)iter.next();
    NodeList child_nodes_of_phy_interface = node.getChildNodes();
    // child_nodes_of_phy_interface contains nodes like <name> and <admin-status>

    // Get each <name> node from the NodeList
    for (int i = 0; i < child_nodes_of_phy_interface.getLength(); i++) {
        Node child_node = child_nodes_of_phy_interface.item(i);
        if (child_node.getNodeType() != Node.ELEMENT_NODE){
            continue;
        }
        if (child_node.getNodeName().equals("name")) {
            // Print the text value of the <name> node
            System.out.println(child_node.getTextContent());
        }
        break;
    }
}
```

**Related
Documentation**

- [Creating and Executing a NETCONF Java Toolkit Program File on page 21](#)
- [NETCONF Java Toolkit Class: Device on page 11](#)
- [NETCONF Java Toolkit Class: NetconfSession on page 13](#)
- [NETCONF Java Toolkit Class: XML on page 15](#)

- [NETCONF Java Toolkit Class: XMLBuilder on page 14](#)
- [NETCONF Java Toolkit Overview on page 4](#)

Troubleshooting Exception Errors in a NETCONF Java Toolkit Program

The following sections outline exception errors that you might encounter when executing a NETCONF Java toolkit program. These sections also present potential causes and solutions for each error.

- [Troubleshooting Connection Errors: Socket Timed Out on page 30](#)
- [Troubleshooting Connection Errors: No Connection on page 30](#)
- [Troubleshooting Authentication Errors on page 31](#)
- [Troubleshooting NETCONF Session Errors on page 31](#)

Troubleshooting Connection Errors: Socket Timed Out

Problem A NETCONF exception occurs, and you see the following error message:

```
Exception in thread "main" net.juniper.netconf.NetconfException: The connect()
operation on the socket timed out.
    at net.juniper.netconf.Device.createNetconfSession(Device.java:344)
    at net.juniper.netconf.Device.connect(Device.java:225)
    at GetChassisInventory.main(GetChassisInventory.java:14)
```

Cause Potential causes for the socket timed out error include:

- The device or interface to which you are connecting is down or unavailable.
- The IP address or DNS name in the arguments for the **Device** object is incorrect.
- The connection timeout value was exceeded before the connection was established.

Solution Ensure that the device is up and running. Also verify that the IP address or DNS name is correct in the arguments of the **Device** constructor in your program code.

The default timeout value for connecting to a device is 5000 milliseconds. To set the timeout value to a larger interval to ensure that the program has sufficient time to establish the connection, call the **setTimeout()** method on the device object. The following code sets the timeout interval to 10 seconds:

```
Device device = new Device("10.10.1.1","admin","PaSsWoRd",null);
device.setTimeout(10000);
device.connect();
```

Troubleshooting Connection Errors: No Connection

Problem An `IllegalStateException` exception occurs, and you see the following error message:

```
Exception in thread "main" java.lang.IllegalStateException: Cannot execute RPC,
you need to establish a connection first.
```

```
at net.juniper.netconf.Device.executeRPC(Device.java:498)
at GetChassisInventoryRun.main(GetChassisInventoryRun.java:15)
```

Cause An SSHv2 connection or NETCONF session was not established with the remote device.

Solution Call the **connect()** method on the device object to establish an SSHv2 connection and a default NETCONF session with the device on which the NETCONF server runs. Once the connection and session are established, RPC execution should be successful.

Troubleshooting Authentication Errors

Problem A NETCONF exception occurs, and you see the following error message:

```
Exception in thread "main" net.juniper.netconf.NetconfException: Authentication
failed.
```

```
at net.juniper.netconf.Device.createNetconfSession(Device.java:358)
at net.juniper.netconf.Device.connect(Device.java:225)
at GetChassisInventory.main(GetChassisInventory.java:14)
```

```
<!-- or -->
```

```
Could not connect to device:Authentication failed.
```

Cause An error message for failed authentication could have several possible causes, including the following:

- The host or authentication details passed as arguments to the **Device** constructor are incorrectly entered in the program code.
- The arguments for the **Device** object are correct, but there is no corresponding user account created on the device to which you are connecting.

Solution If there is no user account on the device to which you are connecting, create the account with the appropriate authentication. For more information about configuring user accounts on a device running Junos OS, see the Junos OS System Basics Configuration Guide.

If the user account exists on the remote device, but the arguments for the **Device** constructor are entered incorrectly in the program code, correct the arguments and recompile the program.

Troubleshooting NETCONF Session Errors

Problem A NETCONF exception occurs, and you see the following error message:

```
Exception in thread "main" net.juniper.netconf.NetconfException: There was a
problem while connecting to 10.10.1.1:830
```

```
at net.juniper.netconf.Device.createNetconfSession(Device.java:344)
at net.juniper.netconf.Device.connect(Device.java:225)
at GetChassisInventory.main(GetChassisInventory.java:14)
```

Cause NETCONF over SSH might not be enabled on the device where the NETCONF server resides, or it might be enabled on a different port.

Solution Ensure that you have enabled NETCONF over SSH on the device where the NETCONF server resides. If your NETCONF Java toolkit program does not specify a specific port number in the **Device** arguments, the NETCONF session is established on the default NETCONF-over-SSH port, 830. To verify whether NETCONF over SSH is enabled on the default port for a device running Junos OS, enter the following operational mode command on the remote device:

```
user@host> show configuration system services
```

```
ftp;
netconf {
    ssh;
}
```

If the **netconf** configuration hierarchy is absent, issue the following statements in configuration mode to enable NETCONF over SSH on the default port:

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

If the **netconf** configuration hierarchy specifies a port other than the default port, you should include the new port number in the **Device** object constructor arguments. For example, the following device is configured for NETCONF over SSH on port 12345:

```
user@host> show configuration system services
netconf {
    ssh {
        port 12345;
    }
}
```

To correct the connection issue, include the new port number in the **Device** arguments.

```
Device device = new Device("10.10.1.1", "admin", "PaSsWoRd", null, 12345);
```

**Related
Documentation**

- [Creating and Executing a NETCONF Java Toolkit Program File on page 21](#)
- [NETCONF Java Toolkit Class: Device on page 11](#)
- [NETCONF Java Toolkit Class: NetconfSession on page 13](#)
- [NETCONF Java Toolkit Overview on page 4](#)

CHAPTER 5

Performing Operational and Configuration Tasks

- [Using the NETCONF Java Toolkit to Perform Operational Tasks on page 33](#)
- [Using the NETCONF Java Toolkit to Perform Configuration Tasks on page 37](#)

Using the NETCONF Java Toolkit to Perform Operational Tasks

- [Using Device Object Methods to Execute RPCs and Operational Commands on page 33](#)
- [Example: Using the NETCONF Java Toolkit to Execute CLI Commands on page 34](#)

Using Device Object Methods to Execute RPCs and Operational Commands

The NETCONF Java toolkit **Device** object has methods to request information from and perform operational tasks on remote devices. When appropriate, the methods are overloaded to take a number of different formats.

- [Executing RPCs on page 33](#)
- [Executing Operational Mode Commands on page 34](#)

Executing RPCs

To execute a remote procedure call (RPC), call the **executeRPC()** method on the **Device** object. The **executeRPC()** method is overloaded to accept a **String** object, a **net.juniper.netconf.XML** object, or an **org.w3c.dom.Document** object as the argument. The RPC is processed by the NETCONF server, which returns the RPC reply as an XML object.

The method syntax is:

```
public XML executeRPC (String rpcContent)
public XML executeRPC (net.juniper.netconf.XML rpc)
public XML executeRPC (org.w3c.dom.Document rpcDoc)
```

The following code snippet executes the Junos XML API **get-chassis-inventory** RPC using a string argument. The **get-chassis-inventory** RPC is equivalent to the **show chassis hardware** operational mode command in the Junos OS command-line interface (CLI).

```
Device device = new Device("10.10.1.1","admin","PaSsWoRd",null);
device.connect();
try {
```

```
XML rpc_reply = device.executeRPC("get-chassis-inventory");
System.out.println(rpc_reply.toString());
}
catch (Exception e) {
    System.out.println("exception: " + e.getMessage());
    // additional processing for exception
}
device.close();
```

Executing Operational Mode Commands

To execute an operational mode command to request information from or perform operational tasks on a device running Junos OS, call the **runCliCommand()** method on the **Device** object. The **runCliCommand()** method sends a Junos OS operational mode command to the NETCONF server on the remote device. The argument is a string representing the operational mode command that you would enter in the Junos OS CLI. The RPC is processed by the NETCONF server, which returns the RPC reply. Starting with Junos OS Release 11.4, the return string is the same ASCII-formatted output that you see in the Junos OS CLI. For devices running earlier versions of Junos OS, the return string contains Junos XML tag elements.

The method syntax is:

```
public String runCLICommand (String command)
```

The following code snippet sends the CLI operational mode command **show chassis hardware** to the NETCONF server on a device running Junos OS:

```
Device device = new Device("10.10.1.1","admin","PaSsWoRd",null);
device.connect();
try {
    cli_reply = device.runCliCommand("show chassis hardware");
    System.out.println(cli_reply);
}
catch (Exception e) {
    System.out.println("exception: " + e.getMessage());
    // additional processing for exception
}
device.close();
```

Example: Using the NETCONF Java Toolkit to Execute CLI Commands

This NETCONF Java toolkit program demonstrates the **runCLICommand()** method, which sends the specified Junos OS operational mode command to the NETCONF server to request information from or perform operational tasks on a device running Junos OS.

- [Requirements on page 34](#)
- [Overview on page 35](#)
- [Configuration on page 35](#)
- [Verification on page 36](#)

Requirements

- Routing, switching, or security device running Junos OS.
- NETCONF Java toolkit is installed on the configuration management server.

- Client application can log in to the device where the NETCONF server resides.
- NETCONF service over SSH is enabled on the device where the NETCONF server resides.

Overview

The NETCONF Java toolkit **Device** class contains the **runCliCommand()** method, which takes a Junos OS CLI operational mode command and converts it to an equivalent RPC in XML that can be processed by the NETCONF server. The **runCliCommand()** method takes as an argument the string representing an operational mode command that you enter in the Junos OS CLI.

The following example executes the **show chassis hardware** command on a device running Junos OS. The return value for the method is a string. Starting with Junos OS Release 11.4, the return string is the same ASCII-formatted output that you see in the Junos OS CLI. For devices running earlier versions of Junos OS, the return string contains Junos XML tag elements.

Configuration

Creating the Java program

Step-by-Step Procedure

To construct the Java program file:

1. Give the file a descriptive name.

The filename must be the same as the class name. For this example, the file and class are named **ExecuteCLICommand**.

2. Add the code to the file and update the environment-specific variables such as the remote host IP address, username, password, and **<rpc-reply>** tag elements.

The complete Java code for the **ExecuteCLICommand.java** program is presented here.

```
/*ExecuteCLICommand*/
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import net.juniper.netconf.Device;
import net.juniper.netconf.NetconfException;
import net.juniper.netconf.XML;
import org.xml.sax.SAXException;

public class ExecuteCLICommand {
    public static void main(String args[]) throws NetconfException,
        ParserConfigurationException, SAXException, IOException {

        String cli = "show chassis hardware";

        Device device = new Device("10.10.1.1","admin","PaSsWoRd",null);
        device.connect();
        try {
            String cli_reply = device.runCliCommand(cli);
            System.out.println(cli_reply);
        }
        catch (Exception e) {
```

```

        System.out.println("exception: " + e.getMessage());
        // additional processing for exception
    }
    device.close();
}
}

```

Compiling and Running the Java Program

Step-by-Step Procedure You need a Java compiler to compile the source code and to create an executable program.

To compile the code and run the program on the configuration management server:

1. Compile the **ExecuteCLICommand.java** file.

```
>javac ExecuteCLICommand.java
```
2. Execute the resulting **ExecuteCLICommand** program.

```
>java ExecuteCLICommand
```

Verification

Verifying Program Execution

Purpose Verify that the **ExecuteCLICommand** program runs correctly.

Action If the program executes successfully, it establishes a connection and creates a NETCONF session with the specified device. The program converts the Junos OS CLI operational mode command **show chassis hardware** to an RPC and sends the RPC to the NETCONF server. The server responds with the requested operational information enclosed in the **<rpc-reply>** tag element. The program parses the RPC reply and prints the resulting chassis inventory. The following sample output is from a Juniper Networks m7i router.

On a device running Junos OS Release 11.4 or later release, the output is in ASCII-formatted text, which is identical to the output in the CLI.

Hardware inventory:

Item	Version	Part number	Serial number	Description
Chassis			30010	M7I
Midplane	REV 03	710-008761	CB3874	M7i Midplane
Power Supply 0	Rev 04	740-008537	PG10715	AC Power Supply
Routing Engine	REV 07	740-009459	1000445584	RE-5.0
CFEB	REV 07	750-010464	CM4612	Internet Processor II
FPC 0				E-FPC
PIC 0	REV 06	750-002971	CB0032	4x OC-3 SONET, MM
PIC 1	REV 02	750-002982	HS2878	1x Tunnel
PIC 2	REV 08	750-005724	CL9084	2x OC-3 ATM-II IQ, MM
PIC 3	REV 12	750-012838	DJ1107	4x 1GE(LAN), IQ2
Xcvr 0	REV 01	740-013111	7303405	SFP-T
Xcvr 1	REV 01	740-013111	7303391	SFP-T
Xcvr 2	REV 01	740-013111	7303350	SFP-T
Xcvr 3	REV 01	740-013111	7303420	SFP-T
FPC 1				E-FPC
PIC 2	REV 07	750-009487	CL5745	ASP - Integrated
(Layer-2-3)				

PIC 3
Fan Tray

REV 07 750-009098 CB7256

2x F/E, 100 BASE-TX
Rear Fan Tray

On a device running Junos OS Release 11.3 or earlier release, the output contains Junos XML tag elements.

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/11.2R1/junos">
  <chassis-inventory xmlns="http://xml.juniper.net/junos/11.2R1/junos-chassis">

    <chassis junos:style="inventory">
      <name>Chassis</name>
      <serial-number>30010</serial-number>
      <description>M7I</description>
      <chassis-module>
        <name>Midplane</name>
        <version>REV 03</version>
        <part-number>710-008761</part-number>
        <serial-number>CB3874</serial-number>
        <description>M7i Midplane</description>
        <model-number>CHAS-MP-M7i-1GE-S</model-number>
      </chassis-module>

      /* Output omitted for brevity */

    </chassis>
  </chassis-inventory>
</rpc-reply>
```

Related Documentation

- [Example: Using the NETCONF Java Toolkit to Execute an Operational Request RPC on page 24](#)
- [Example: Using the NETCONF Java Toolkit to Load and Commit a Configuration on page 40](#)
- [Troubleshooting Exception Errors in a NETCONF Java Toolkit Program on page 30](#)
- [NETCONF Java Toolkit Class: Device on page 11](#)
- [NETCONF Java Toolkit Overview on page 4](#)

Using the NETCONF Java Toolkit to Perform Configuration Tasks

- [Using Device Object Methods to Load Configuration Changes on page 37](#)
- [Example: Using the NETCONF Java Toolkit to Load and Commit a Configuration on page 40](#)
- [Example: Using the NETCONF Java Toolkit to Load Set Configuration Commands on page 43](#)

Using Device Object Methods to Load Configuration Changes

The NETCONF Java toolkit **Device** object has methods to help you configure remote devices. When appropriate, the methods are overloaded to take a number of different formats.

To load configuration data on a remote device, the **Device** object has several methods that enable you to define the configuration data as a set of Junos OS configuration mode commands, formatted ASCII text, or Junos XML tag elements. You can supply the configuration data in the program code, or you can reference data files that include the desired configuration changes.

To configure a private copy of the candidate configuration, call the **openConfiguration("private")** method with the string argument "private" on the device object before loading your configuration changes. This is equivalent to the **configure private** command in the Junos OS CLI. If you omit the call to the **openConfiguration("private")** method, your configuration changes are loaded into the global copy of the candidate configuration.

The method used to load the configuration data depends on the source and the format of the data. In the following methods, the string argument **loadType** has a value of either **merge** or **replace**, which performs the equivalent of the configuration mode commands **load merge** or **load replace** on a device running Junos OS.

- *Junos OS configuration mode commands*—The following methods load configuration data as a set of Junos OS configuration mode commands. These methods are only supported on devices running Junos OS Release 11.4 or a later release. Junos OS executes the configuration instructions line by line. For each element, you can specify the complete statement path in the command, or you can use navigation commands, such as **edit**, to move around the configuration hierarchy as you do in CLI configuration mode.
 - **loadSetConfiguration(String setCommands)**—Specify the configuration data in the program code, either as a method argument or as a variable passed to the method.
 - **loadSetFile(String filePath)**—Load the configuration data from the file specified by *filePath*.
- *Formatted ASCII text*—The following methods load configuration data as formatted ASCII text. Use the standard Junos OS CLI notations—the newline character, tabs, spaces, braces, and square brackets—to indicate the hierarchical relationships between configuration statements.
 - **loadTextConfiguration(String textConfiguration, String loadType)**—Specify the configuration data in the program code, either as a method argument or as a variable passed to the method.
 - **loadTextFile(String filePath, String loadType)**—Load the configuration data from the file specified by *filePath*.
- *Junos XML tag elements*—The following methods load configuration data as Junos XML tag elements. Include the tag elements representing all levels of the configuration hierarchy under the root, the **<configuration>** tag element, down to each new or changed element.
 - **loadXMLConfiguration(String XMLConfiguration, String loadType)**—Specify the configuration data in the program code as a **net.juniper.netconf.XML** object, which is passed to the method.

- `loadXMLFile(String filePath, String loadType)`—Load the configuration data from the file specified by `filePath`.

The following code snippet merges the `ftp` statement into the candidate configuration at the `[edit system services]` hierarchy level. The Java statement for each type of load configuration method is shown. When loading from a file, the file should contain the appropriate hierarchy in the desired format.

```
/*
r1-config-set.txt:
set system services ftp

r1-config-text.txt:
system {
  services {
    ftp;
  }
}

r1-config-xml.txt:
<system>
  <services>
    <ftp/>
  </services>
</system>
*/

String config_file_set = "configs/r1-config-set.txt"
String config_file_text = "configs/r1-config-text.txt"
String config_file_xml = "configs/r1-config-xml.txt"

XMLBuilder builder = new XMLBuilder();
XML ftp_config = builder.createNewConfig("system", "services", "ftp");

Device device = new Device("10.10.1.1", "admin", "PaSsWoRd", null);
device.connect();

//open a private copy of the candidate configuration
device.openConfiguration("private");

// load configuration data as Junos OS configuration mode commands
device.loadSetConfiguration("set system services ftp");
device.loadSetFile(config_file_set);

// load configuration data as formatted ASCII text
device.loadTextConfiguration("system { services { ftp; } }", "merge");
device.loadTextFile(config_file_text, "merge");

// load configuration data as Junos XML tag elements
device.loadXMLConfiguration(ftp_config.toString(), "merge");
device.loadXMLFile(config_file_xml, "merge");
```

```
device.commit();  
device.close();
```

Example: Using the NETCONF Java Toolkit to Load and Commit a Configuration

The following example NETCONF Java toolkit program constructs a configuration hierarchy, which is then merged with the candidate configuration on the specified device. The resulting configuration is then committed. The sample configuration hierarchy is for a device running Junos OS.

- [Requirements on page 40](#)
- [Overview on page 40](#)
- [Configuration on page 41](#)
- [Verification on page 42](#)
- [Troubleshooting on page 43](#)

Requirements

- Routing, switching, or security device running Junos OS.
- NETCONF Java toolkit is installed on the configuration management server.
- Client application can log in to the device where the NETCONF server resides.
- NETCONF service over SSH is enabled on the device where the NETCONF server resides.

Overview

The following example performs a **load merge** operation to update the candidate configuration on a device running Junos OS and then commits the new configuration. The XML hierarchy that will be added into the configuration is constructed with the **XMLBuilder** object and stored in the **ftp_config** variable. Alternatively, you can load configuration data as text and, for devices running Junos OS Release 11.4 or a later release, as a set of Junos OS configuration mode commands.

The new configuration hierarchy, which enables FTP service on the device, is:

```
<configuration>  
  <system>  
    <services>  
      <ftp/>  
    </services>  
  </system>  
</configuration>
```

The program code creates a new **Device** object and calls the **connect()** method. This establishes an SSHv2 connection and a default NETCONF session with the device on which the NETCONF server runs.

To prevent conflicts with other users who might simultaneously edit the candidate configuration, the code calls the **lockConfig()** method on the device object to lock the configuration. If the lock fails, the method generates an error message, and the program exits. If the lock is successful, the **loadXMLConfiguration(ftp_config.toString(), "merge")**

method loads the new configuration hierarchy into the candidate configuration using the **merge** option. Notice that, although the configuration hierarchy is initially constructed as XML, you must convert it to a string before passing it as an argument to the **loadXMLConfiguration()** method.

Once the new configuration hierarchy is merged with the candidate configuration, the program attempts to commit the configuration. If the commit operation is unsuccessful, the program prints the associated error message. The program then unlocks the configuration and closes the NETCONF session and device connection.



NOTE: For more information about the **merge** and **replace** options for loading configuration hierarchies and statements into the candidate configuration, see the CLI User Guide.

Configuration

Creating the Java Program

Step-by-Step Procedure

To construct the Java program file that contains the code for the configuration changes and requests:

1. Give the file a descriptive name.

The filename must be the same as the class name. For this example, the file and class are named **EditConfig**.

2. Add the code to the file and update the environment-specific variables such as the remote host IP address, username, and password.

The complete Java code for the EditConfig program is presented here.

```
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import net.juniper.netconf.CommitException;
import net.juniper.netconf.Device;
import net.juniper.netconf.LoadException;
import net.juniper.netconf.NetconfException;
import net.juniper.netconf.XML;
import net.juniper.netconf.XMLBuilder;
import org.xml.sax.SAXException;

public class EditConfig {
    public static void main(String[] args) throws LoadException, IOException,
        NetconfException, ParserConfigurationException, SAXException {

        /*Build the following XML hierarchy to add to the configuration:
        * <configuration>
        *   <system>
        *     <services>
        *       <ftp/>
        *     </services>
        *   </system>
        * </configuration>
```

```
*/

XMLBuilder builder = new XMLBuilder();
XML ftp_config = builder.createNewConfig("system", "services", "ftp");

//Create the device
Device device = new Device("10.10.1.1","admin","PaSsWoRd",null);
device.connect();

//Lock the configuration
boolean isLocked = device.lockConfig();
if(!isLocked) {
    System.out.println("Could not lock configuration. Exit now.");
    return;
}

//Load and commit the configuration
try {
    device.loadXMLConfiguration(ftp_config.toString(), "merge");
    device.commit();
} catch(LoadException e) {
    System.out.println(e.getMessage());
    return;
} catch(CommitException e) {
    System.out.println(e.getMessage());
    return;
}

//Unlock the configuration and close the device
device.unlockConfig();
device.close();
}
}
```

Compiling and Running the Java Program

Step-by-Step Procedure You need a Java compiler to compile the source code and to create an executable program.

To compile the code and run the program on the configuration management server:

1. Compile the **EditConfig.java** file.

```
>javac EditConfig.java
```
2. Execute the resulting **EditConfig** program.

```
>java EditConfig
```

Verification

Verifying Program Execution

Purpose Verify that the **EditConfig** program runs correctly.

Action If the program executes successfully, it establishes a connection and creates a NETCONF session with the specified device. The program merges the new hierarchy with the candidate configuration on the device and commits the configuration.

You can verify that the configuration was correctly merged and committed by viewing the resulting configuration on the remote device. The **ftp** statement should now be in the active configuration. On a device running Junos OS, enter the following operational mode command to view the **[edit system services]** hierarchy:

```
user@host> show configuration system services
ftp;
netconf {
    ssh;
}
```

Troubleshooting

- [Troubleshooting Error Messages on page 43](#)

Troubleshooting Error Messages

Problem The following error message is printed to the display:

```
Could not lock configuration. Exit now.
```

Solution Another user currently has a lock on the candidate configuration. Wait until the lock is released and execute the program.

Example: Using the NETCONF Java Toolkit to Load Set Configuration Commands

This NETCONF Java toolkit program demonstrates the **loadSetConfiguration()** method, which updates the configuration using a set of Junos OS configuration mode commands.

- [Requirements on page 43](#)
- [Overview on page 43](#)
- [Configuration on page 44](#)
- [Verification on page 46](#)

Requirements

- Routing, switching, or security device running Junos OS Release 11.4 or later.
- NETCONF Java toolkit is installed on the configuration management server.
- Client application can log in to the device where the NETCONF server resides.
- NETCONF service over SSH is enabled on the device where the NETCONF server resides.

Overview

The **Device** class contains the **loadSetConfiguration()** and **loadSetFile()** methods, which load configuration data as a set of Junos OS configuration mode commands on devices running Junos OS Release 11.4 or a later release. For each configuration element, you can specify the complete statement path in the command, or you can use navigation

commands, such as **edit**, to move around the configuration hierarchy as you do in CLI configuration mode. The NETCONF Java toolkit converts the command set to the equivalent RPC in XML that can be processed by the NETCONF server on devices running Junos OS. Junos OS executes the configuration instructions line by line.

The method syntax is:

```
public void loadSetConfiguration (String setCommands)
public void loadSetFile (String filePath)
```

The **loadSetConfiguration()** method takes as an argument the configuration command string that you would enter in Junos OS CLI configuration mode. For example, to add the **ftp** statement at the **[edit system services]** hierarchy level, you use the **set system services ftp** command. The **loadSetFile()** method takes as an argument the path of the file containing the set of configuration commands.

You can also use both methods to load multiple commands. To load multiple commands using the **loadSetConfiguration()** method, you can either list the commands as a single string and separate them with the `\n` newline sequence, or you can execute the method separately for each command. To load multiple commands using the **loadSetFile()** method, place each command on a separate line in the file.



NOTE: When using the **loadSetConfiguration()** method with navigation commands, you should list the commands as a single string and separate them with the `\n` newline sequence. You cannot call the **loadSetConfiguration()** method with a single navigation command such as **up**.

The program in this example loads two configuration commands, which merge two statements into the candidate configuration on a device running Junos OS Release 11.4. The first command, **set system services ftp**, adds the **ftp** statement at the **[edit system services]** hierarchy level. The second command, **set interfaces ge-0/0/0 disable**, adds the **disable** statement at the **[edit interfaces ge-0/0/0]** hierarchy level. The relevant statements in the program code are:

```
String system_config = "set system services ftp";
String interfaces_config = "set interfaces ge-0/0/0 disable";

device.loadSetConfiguration(system_config);
device.loadSetConfiguration(interfaces_config);
```

Configuration

Creating the Java Program

Step-by-Step Procedure

To construct the Java program file:

1. Give the file a descriptive name.

The filename must be the same as the class name. For this example, the file and class are named **LoadSetConfig**.
2. Add the code to the file and update the environment-specific variables such as the remote host IP address, username, password, and **<rpc-reply>** tag elements.

The complete Java code for the **LoadSetConfig.java** program is presented here.

If you load the set of commands from a file, create a file containing the commands, and replace the two **loadSetConfiguration()** method calls with a call to the **loadSetFile()** method.

```

/*LoadSetConfig*/
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import net.juniper.netconf.Device;
import net.juniper.netconf.CommitException;
import net.juniper.netconf.LoadException;
import net.juniper.netconf.NetconfException;
import net.juniper.netconf.XML;
import org.xml.sax.SAXException;

public class LoadSetConfig {
    public static void main(String args[]) throws NetconfException,
        ParserConfigurationException, SAXException, IOException {

        String system_config = "set system services ftp";
        String interfaces_config = "set interfaces ge-0/0/0 disable";

        Device device = new Device("10.10.1.1","admin","PaSsWoRd",null);

        try {
            device.connect();
            System.out.println("Connection successful.");

            if (device.lockConfig()) {
                System.out.println("Configuration successfully locked.");
                try {
                    System.out.println("Updating configuration.");
                    device.loadSetConfiguration(system_config);
                    device.loadSetConfiguration(interfaces_config);
                    System.out.println("Committing configuration.");
                    device.commit();
                }
                catch (LoadException e) {
                    System.out.println("LoadException occurred: " + e.getMessage());
                }
                catch (CommitException e) {
                    System.out.println("CommitException occurred: " + e.getMessage());
                }
                device.unlockConfig();
                device.close();
            }
            else {
                System.out.println("Error - cannot lock configuration");
            }
        }
        catch (NetconfException e) {
            System.out.println("Could not connect to device: " + e.getMessage());
        }
    }
}

```

Compiling and Running the Java Program

Step-by-Step Procedure You need a Java compiler to compile the source code and to create an executable program.

To compile the code and run the program on the configuration management server:

1. Compile the **LoadSetConfig.java** file.

```
>javac LoadSetConfig.java
```
2. Execute the resulting **LoadSetConfig** program.

```
>java LoadSetConfig
```

Verification

To confirm that the program is working properly:

- [Verifying Program Execution on page 46](#)
- [Verifying the Configuration Changes on page 46](#)
- [Verifying the Commit on page 47](#)

Verifying Program Execution

Purpose Verify that the **LoadSetConfig** program runs correctly.

Action If the program executes successfully, it establishes a connection and creates a NETCONF session with the specified device. The program merges the new statements with the candidate configuration on the device and commits the configuration.

```
>java LoadSetConfig
Connection successful.
Configuration successfully locked.
Updating configuration.
Committing configuration.
```

Verifying the Configuration Changes

Purpose You can verify that the configuration was correctly merged and committed by viewing the resulting configuration on the remote device. The **ftp** and the **disable** statements should now be in the active configuration. On a device running Junos OS, issue the following operational mode commands to view the **[edit system services]** and **[edit interfaces]** hierarchy levels:

Action

```
admin@host> show configuration system services
ftp;
netconf {
    ssh;
}

admin@host> show configuration interfaces
ge-0/0/0 {
    disable;
}
```

Verifying the Commit

Purpose Additionally, you can review the commit log to verify that the commit was successful. On a device running Junos OS, issue the **show system commit** operational mode command to view the commit log. In this example, the log confirms that the user **admin** committed the candidate configuration in a NETCONF session at the given date and time.

Action Issue the **show system commit** operational mode command and review the commit log.

```
admin@host> show system commit
0   2011-09-02 14:16:44 PDT by admin via netconf
1   2011-07-08 14:33:46 PDT by root via other
```

- Related Documentation**
- [Example: Using the NETCONF Java Toolkit to Execute an Operational Request RPC on page 24](#)
 - [Example: Using the NETCONF Java Toolkit to Load and Commit a Configuration on page 40](#)
 - [Troubleshooting Exception Errors in a NETCONF Java Toolkit Program on page 30](#)
 - [NETCONF Java Toolkit Class: Device on page 11](#)
 - [NETCONF Java Toolkit Overview on page 4](#)

CHAPTER 6

NETCONF Java Toolkit Examples

- [Example: Using the NETCONF Java Toolkit to Execute an Operational Request RPC on page 49](#)
- [Example: Using the NETCONF Java Toolkit to Print Component Temperatures on page 53](#)

Example: Using the NETCONF Java Toolkit to Execute an Operational Request RPC

This NETCONF Java toolkit program executes an RPC to obtain operational information from a device, which is then printed to standard output. This example serves as an instructional example for creating and executing a basic NETCONF Java toolkit program.

- [Requirements on page 49](#)
- [Overview on page 49](#)
- [Configuration on page 50](#)
- [Verification on page 51](#)
- [Troubleshooting on page 52](#)

Requirements

- NETCONF Java toolkit is installed on the configuration management server.
- Client application can log in to the device where the NETCONF server resides.
- NETCONF service over SSH is enabled on the device where the NETCONF server resides.

Overview

You can use the NETCONF Java toolkit to request operational information from a remote device. The following example illustrates how to create a NETCONF Java toolkit program to execute an operational request from the Junos XML API on a device running Junos OS. The example also explains how to compile the code, execute the program, and verify the results.

Configuration

Creating the Java Program

Step-by-Step Procedure

To construct the Java program file that contains the code for the operational request:

1. Give the file a descriptive name.

The filename must be the same as the class name. For this example, the file and class are named **GetChassisInventory**.

2. Include the appropriate import statements, and the code for the class declaration and the Java method, **main()**.

```
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import net.juniper.netconf.Device;
import net.juniper.netconf.NetconfException;
import net.juniper.netconf.XML;
import org.xml.sax.SAXException;

public class GetChassisInventory {
    public static void main(String args[]) throws NetconfException,
        ParserConfigurationException, SAXException, IOException {
    }
}
```

3. Within **main()**, create a **Device** object and call the **connect()** method.

This creates a default NETCONF session over SSHv2 with the NETCONF server. You must update the code with the appropriate arguments for connection to and authentication on your specific device.

```
Device device = new Device("10.10.1.1", "admin", "PaSsWoRd", null);
device.connect();
```

Having established a **Device** object, you can perform NETCONF operations on the device. For a complete list of available methods corresponding to NETCONF operations, refer to the NETCONF Java toolkit Javadocs.

4. Call the **executeRPC()** method with the operational request RPC command as the argument.

This example uses the Junos XML API **get-chassis-inventory** RPC. The reply, which is returned in XML, is stored in the **rpc_reply** variable.

```
XML rpc_reply = device.executeRPC("get-chassis-inventory");
```

5. Add code to take action on the RPC reply.

The following code converts the NETCONF server's reply to a string and prints it to the screen:

```
System.out.println(rpc_reply.toString());
```

6. Close the device and release resources by calling the **close()** method on the device object.

```
device.close();
```


Results The complete program is:

```

/*GetChassisInventory*/
import java.io.IOException;
import javax.xml.parsers.ParserConfigurationException;
import net.juniper.netconf.Device;
import net.juniper.netconf.NetconfException;
import net.juniper.netconf.XML;
import org.xml.sax.SAXException;

public class GetChassisInventory {
    public static void main(String args[]) throws NetconfException,
        ParserConfigurationException, SAXException, IOException {

        Device device = new Device("10.10.1.1","admin","PaSsWoRd",null);
        device.connect();
        XML rpc_reply = device.executeRPC("get-chassis-inventory");
        System.out.println(rpc_reply.toString());
        device.close();
    }
}

```

Compiling and Running the Java Program

Step-by-Step Procedure You need a Java compiler to compile the source code and to create an executable program.

To compile the code and run the program on the configuration management server:

1. Compile the **GetChassisInventory.java** file.
`>javac GetChassisInventory.java`
2. Execute the resulting **GetChassisInventory** program.
`>java GetChassisInventory`

Verification

Verifying Program Execution

Purpose Verify that the GetChassisInventory program runs correctly.

Action If the program executes successfully, it establishes a connection and creates a NETCONF session with the specified device. The program sends the **get-chassis-inventory** RPC to the NETCONF server, and the server responds with the requested operational information enclosed in the **<rpc-reply>** tag element. The program prints the reply to standard out. Following is a sample RPC reply with some output omitted for brevity.

```

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
  xmlns:junos="http://xml.juniper.net/junos/11.2R1/junos">
  <chassis-inventory xmlns="http://xml.juniper.net/junos/11.2R1/junos-chassis">
  <chassis junos:style="inventory">
  <name>Chassis</name>
  <serial-number>12345</serial-number>

```

```
<description>M7i</description>
<chassis-module>

...output omitted...

</chassis>
</chassis-inventory>
</rpc-reply>
```

Troubleshooting

- [Troubleshooting NETCONF Exceptions on page 52](#)

Troubleshooting NETCONF Exceptions

Problem A NETCONF exception occurs, and you see the following error message:

```
Exception in thread "main" net.juniper.netconf.NetconfException: There was a
problem while connecting to 10.10.1.1:830
    at net.juniper.netconf.Device.createNetconfSession(Device.java:344)
    at net.juniper.netconf.Device.connect(Device.java:225)
    at GetChassisInventory.main(GetChassisInventory.java:14)
```

NETCONF over SSH might not be enabled on the device where the NETCONF server resides, or it might be enabled on a different port.

Solution Ensure that you have enabled NETCONF over SSH on the device where the NETCONF server resides. Since the example program does not specify a specific port number in the **Device** arguments, the NETCONF session is established on the default NETCONF-over-SSH port, 830. To verify whether NETCONF over SSH is enabled on the default port for a device running Junos OS, enter the following operational mode command on the remote device:

```
user@host> show configuration system services
```

```
ftp;
netconf {
    ssh;
}
```

If the **netconf** configuration hierarchy is absent, issue the following statements in configuration mode to enable NETCONF over SSH on the default port:

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

If the **netconf** configuration hierarchy specifies a port other than the default port, include the new port number in the **Device** object constructor arguments. For example, the following device is configured for NETCONF over SSH on port 12345:

```
user@host> show configuration system services
netconf {
    ssh {
        port 12345;
    }
}
```

To correct the connection issue, include the new port number in the **Device** arguments.

```
Device device = new Device("10.10.1.1", "admin", "PaSsWoRd", null, 12345);
```

Related Documentation

- [Example: Using the NETCONF Java Toolkit to Execute CLI Commands on page 34](#)
- [Example: Using the NETCONF Java Toolkit to Load and Commit a Configuration on page 40](#)
- [Example: Using the NETCONF Java Toolkit to Load Set Configuration Commands on page 43](#)
- [Example: Using the NETCONF Java Toolkit to Print Component Temperatures on page 53](#)
- [Troubleshooting Exception Errors in a NETCONF Java Toolkit Program on page 30](#)
- [NETCONF Java Toolkit Overview on page 4](#)

Example: Using the NETCONF Java Toolkit to Print Component Temperatures

This NETCONF Java toolkit program prints the name and corresponding temperature of components on a device running Junos OS.

- [Requirements on page 53](#)
- [Overview on page 53](#)
- [Configuration on page 54](#)
- [Verification on page 55](#)

Requirements

- Routing, switching, or security device running Junos OS.
- NETCONF Java toolkit is installed on the configuration management server.
- Client application can log in to the device where the NETCONF server resides.
- NETCONF service over SSH is enabled on the device where the NETCONF server resides.

Overview

The following example executes the Junos XML API **get-environment-information** RPC, which is the equivalent of the **show chassis environment** operational mode command on a device running Junos OS. The program parses the RPC reply, and for all components that list a temperature, the program prints the component name and corresponding temperature.

The RPC reply format for the **get-environment-information** RPC request is:

```
<rpc-reply>
  <environment-information>
    <environment-item>
      <name>item-name</name>
```

```
...
    <temperature>temperature</temperature>
</environment-item>
<environment-item>
    <name>item-name2</name>
    ...
    <temperature>temperature</temperature>
</environment-item>
...
</environment-information>
</rpc-reply>
```

To parse the reply, the program uses the `findNodes()` method to return a list of `org.w3c.dom.Node` objects. For each `<environment-item>` node, the program obtains a list of child nodes. If a temperature element is present in the child node list, the program prints the name and temperature of that environment item.

Configuration

Creating the Java program

Step-by-Step Procedure

To construct the Java program file:

1. Give the file a descriptive name.

The filename must be the same as the class name. For this example, the file and class are named **ShowTemps**.

2. Add the code to the file and update the environment-specific variables such as the remote host IP address, username, password, and `<rpc-reply>` tag elements.

The complete Java code for the **ShowTemps.java** program is presented here.

```
import java.io.IOException;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import javax.xml.parsers.ParserConfigurationException;
import net.juniper.netconf.CommitException;
import net.juniper.netconf.Device;
import net.juniper.netconf.LoadException;
import net.juniper.netconf.NetconfException;
import net.juniper.netconf.XML;
import net.juniper.netconf.XMLBuilder;
import org.xml.sax.SAXException;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class showTemps {
    public static void main(String[] args) throws LoadException, IOException,
        NetconfException, ParserConfigurationException, SAXException {

        String name="", temp="";

        //Create the device
        Device device = new Device("10.10.1.1","admin","PaSsWoRd",null);
        device.connect();
```

```

//Call executeRPC(String rpc) to send RPC and receive RPC reply
XML rpc_reply = device.executeRPC("get-environment-information");

// Parse reply and only print items that have a temperature element
List<String> list =
    Arrays.asList("environment-information","environment-item");
List itemlist = rpc_reply.findNodes(list);
Iterator iter = itemlist.iterator();

while (iter.hasNext()) {
    Node item_node = (Node) iter.next();
    NodeList child_nodes = item_node.getChildNodes();
    // child_nodes contains nodes like <name> and <temperature>

    for (int i = 0; i < child_nodes.getLength(); i++) {
        Node child = child_nodes.item(i);
        if (child.getNodeType() == Node.ELEMENT_NODE) {

            if (child.getNodeName().equals("name"))
                // Capture the text value in <name> node
                name = child.getTextContent();
            if (child.getNodeName().equals("temperature")) {
                // Capture the text value in <temperature> node
                temp = child.getTextContent();
                System.out.println(name + ": " + temp);
            }
        }
    }
}

device.close();
}
}

```

Compiling and Running the Java Program

Step-by-Step Procedure

You need a Java compiler to compile the source code and to create an executable program.

To compile the code and run the program on the configuration management server:

1. Compile the **ShowTemps.java** file.

```
>javac ShowTemps.java
```
2. Execute the resulting **ShowTemps** program.

```
>java ShowTemps
```

Verification

Verifying the Results

Purpose Verify that the **ShowTemps** program runs correctly.

Action If the program executes successfully, it establishes a connection and creates a NETCONF session with the specified device. The program then executes the Junos XML API **get-environment-information** RPC, parses the RPC reply, and prints all environment items that contain a child node **<temperature>**.

The following sample output is from a Juniper Networks m7i router:

```
Intake: 25 degrees C / 77 degrees F
FPC 0: 26 degrees C / 78 degrees F
Power Supplies: 28 degrees C / 82 degrees F
CFEB Intake: 22 degrees C / 71 degrees F
CFEB Exhaust: 30 degrees C / 86 degrees F
Routing Engine: 28 degrees C / 82 degrees F
Routing Engine CPU: 28 degrees C / 82 degrees F
```

**Related
Documentation**

- [Example: Using the NETCONF Java Toolkit to Execute an Operational Request RPC on page 24](#)
- [Example: Using the NETCONF Java Toolkit to Execute CLI Commands on page 34](#)
- [Example: Using the NETCONF Java Toolkit to Load and Commit a Configuration on page 40](#)
- [Example: Using the NETCONF Java Toolkit to Load Set Configuration Commands on page 43](#)
- [Troubleshooting Exception Errors in a NETCONF Java Toolkit Program on page 30](#)
- [Using the NETCONF Java Toolkit to Parse an RPC Reply on page 27](#)
- [NETCONF Java Toolkit Overview on page 4](#)

PART 3

Index

- [Index on page 59](#)

Index

Symbols

#, comments in configuration statements.....	xi
(), in syntax descriptions.....	xi
< >, in syntax descriptions.....	xi
[], in configuration statements.....	xi
{ }, in configuration statements.....	xi
(pipe), in syntax descriptions.....	xi

B

braces, in configuration statements.....	xi
brackets	
angle, in syntax descriptions.....	xi
square, in configuration statements.....	xi

C

Class	
Device.....	11, 33, 37
NetconfSession.....	13
XML.....	15
XMLBuilder.....	14
comments, in configuration statements.....	xi
compiling NETCONF Java toolkit program files.....	23
configuration	
committing.....	40
locking.....	40
requesting changes.....	37, 40, 43
conventions	
text and syntax.....	xi
creating NETCONF Java toolkit program files.....	22
curly braces, in configuration statements.....	xi
customer support.....	xii
contacting JTAC.....	xii

D

Device class.....	11, 33, 37
methods.....	33, 37
documentation	
comments on.....	xii
downloading the NETCONF Java toolkit.....	7

E

errors	
exceptions.....	30
troubleshooting.....	30
examples of NETCONF Java toolkit programs	
executing CLI commands.....	34
executing configuration mode commands.....	43
executing operational requests.....	24, 49
loading and committing a configuration.....	40
printing component temperatures.....	53
exception errors.....	30
authentication error.....	31
connection error.....	30
NETCONF session error.....	31
socket timed out error.....	30
executeRPC method.....	24, 33, 49
executing NETCONF Java toolkit program files.....	23

F

findNodes method.....	27, 54
findValue method.....	27
font conventions.....	xi

I

icons defined, notice.....	x
IllegalStateException	
connection error.....	30
installing the NETCONF Java toolkit.....	7

J

Java program files.....	21
compiling.....	23
creating.....	22
executing.....	23
Junos XML API	
overview.....	3

L

loadSetConfiguration method.....	37, 43
loadSetFile method.....	37
loadTextConfiguration method.....	37
loadTextFile method.....	37
loadXMLConfiguration method.....	37, 40
loadXMLFile method.....	37

M

manuals	
comments on.....	xii

methods

createNewConfig.....	14
createNewRPC.....	14
createNewXML.....	14
executeRPC.....	24, 33, 49
findNodes.....	27, 54
findValue.....	27
loadSetConfiguration.....	37, 43
loadSetFile.....	37
loadTextConfiguration.....	37
loadTextFile.....	37
loadXMLConfiguration.....	37, 40
loadXMLFile.....	37
runCliCommand.....	34

N

NETCONF Java toolkit

classes.....	4, 11, 13, 14, 15
compiling Java program files.....	23
creating Java program files.....	22
Device class.....	11, 33, 37
downloading.....	7
examples.....	24, 34, 40, 43, 49, 53
exception errors.....	30
executing Java program files.....	23
installing.....	7
NetconfSession class.....	13
overview.....	4
parsing a reply.....	27
program files.....	21
troubleshooting.....	30
XML class.....	15
XMLBuilder class.....	14

NETCONF session requirements.....8

NETCONF XML management protocol

overview.....	3
server See NETCONF XML protocol server	

NETCONF XML protocol server.....3

NetconfException

authentication error.....	31
NETCONF session error.....	31
socket timed out error.....	30

NetconfSession class.....13

notice icons defined.....x

O

operational mode commands

executing.....	34
----------------	----

operational requests

executing an operational mode command.....	34
executing an RPC.....	24, 33, 49

P

parentheses, in syntax descriptions.....xi

parsing RPC replies from the NETCONF server.....27

R

RPC

executing.....	24, 33, 49
parsing a reply.....	27

RPC reply

parsing.....	27
--------------	----

runCliCommand method.....34, 43

S

server See NETCONF XML protocol server

session, NETCONF

requirements.....	8
-------------------	---

SSH connection.....8

support, technical See technical support

syntax conventions.....xi

T

technical support

contacting JTAC.....	xii
----------------------	-----

troubleshooting exception errors.....30

X

XML class.....15

XMLBuilder class.....14