JUNIPER NETWORKS

Junos® OS

Configuration and Operations Automation

Release
11.4

Published: 2011-11-08
Revision 1

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

Junos® OS Configuration and Operations Automation

Revision History
October 2011—Revision 1; initial release

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

END USER LICENSE AGREEMENT

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

Table of Contents

List of Figures

List of Tables

PART 1

Overview

CHAPTER 1

Introduction to Junos Automation

Junos Automation Overview

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

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

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

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

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

Junos Configuration Automation: Commit Scripts

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

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

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

Junos Operations Automation: Op Scripts

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

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

Junos Event Automation: Event Scripts and Event Policy

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

Event Policy

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

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

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

Event Scripts

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

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

Related Documentation

- Commit Script Overview

CHAPTER 2

Junos XML API and Junos XML Management Protocol

Junos XML API and Junos XML Management Protocol Overview

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

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

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

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

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

Related Documentation

- Advantages of Using the Junos XML Management Protocol and Junos XML API on page 8
- XML and Junos OS on page 49
- XML Overview on page 47

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

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

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

Parsing Device Output

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

The formatted ASCII follows:

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

The corresponding XML-tagged version is:

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

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

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

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

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

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

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

Displaying Device Output

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

Related Documentation

- Junos XML API and Junos XML Management Protocol Overview on page 7
- XML Overview on page 47

CHAPTER 3

Extension Functions, Templates, and Parameters in the jcs Namespace

Junos Script Automation: Extension Functions in the jcs Namespace Overview

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

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

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

The following example maps the **jcs** prefix to the namespace identified by the URI http://xml.juniper.net/junos/commit-scripts/1.0, which defines the extension functions used in commit, op, and event scripts.The script then calls the **jcs:invoke()** function with one argument.

XSLT Syntax

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
    ...
    <xsl:variable name="result" select="jcs:invoke($command)"/>
    ...
</xsl: stylesheet>
```

SLAX Syntax

```
version 1.0;
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
    ...
    var $result = jcs:invoke($command);
    ...
```

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 12
- Junos Script Automation: Named Templates in the jcs Namespace Overview on page 32
- Junos Script Automation: Global Parameters and Variables in the junos.xsl File on page 42
- SLAX Variables Overview on page 84
- XSLT Variables Overview on page 63

Junos Extension Functions in the jcs Namespace Summary

The Junos extension functions are summarized in the following table.

Table 1: Junos Extension Functions

Function	Type	Description
jcs:break-lines()	Data manipulation	Break a simple element into multiple elements, delimited by newlines.
jcs:close()	Utility	Close a previously opened connection handle.
jcs:dampen()	Utility	Prevent the same operation from being repeatedly executed within a script.
jcs:empty()	Logic	Evaluate a node set or string argument to determine if it is an empty value.
jcs:execute()	Utility	Execute a remote procedure call (RPC) within the context of a specified connection handle.

Table 1: Junos Extension Functions *(continued)*

Function	Type	Description
jcs:first-of()	Logic	Return the first nonempty (non-null) item in a list, If all objects in the list are empty, the default expression is returned.
jcs:get-hello()	Utility	Return the session ID and the capabilities of the NETCONF server during a NETCONF session.
jcs:get-input()	Input/output control	Invoke a CLI prompt and wait for user input. If the script is run non-interactively, the function returns an empty value. This function cannot be used with event scripts.
jcs:get-protocol()	Utility	Return the session protocol associated with the connection handle.
jcs:get-secret()	Input/output control	Invoke a CLI prompt and wait for user input. The input is not echoed back to the user.
jcs:hostname()	Utility	Return the fully qualified domain name associated with a given IPv4 or IPv6 address, provided the DNS server is configured on the router.
jcs:invoke()	Utility	Invoke an RPC on the local device.
jcs:open()	Utility	Return a connection handle that can be used to execute RPCs.
jcs:output()	Input/output control	Generate unformatted output text that is immediately sent to the CLI session.
jcs:parse-ip()	Data manipulation	Parse an IPv4 or IPv6 address and return the host IP address, protocol family, prefix length, network address, and network mask.
jcs:printf()	Input/output control	Generate formatted output text. Most standard **printf** formats are supported, in addition to some Junos OS-specific formats. The function returns a formatted string but does not print it on call.
jcs:progress()	Input/output control	Issue a progress message containing the single argument immediately to the CLI session provided that the **detail** flag was specified when the script was invoked.
jcs:regex()	Data manipulation	Evaluate a regular expression against a given string argument and return any matches.
jcs:sleep()	Utility	Cause the script to sleep for a specified time.
jcs:split()	Data manipulation	Split a string into an array of substrings delimited by a regular expression pattern.

Table 1: Junos Extension Functions *(continued)*

Function	Type	Description
jcs:sysctl()	Utility	Return the value of the given **sysctl** value as a string or an integer.
jcs:syslog()	Input/output control	Log messages with the specified priority to the system log file.
jcs:trace()	Input/output control	Issue a trace message, which is sent to the trace file.

Related Documentation

- Junos Named Templates in the jcs Namespace Summary on page 33
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 11
- Junos Script Automation: Named Templates in the jcs Namespace Overview on page 32
- Junos Script Automation: Global Parameters and Variables in the junos.xsl File on page 42

Junos Extension Functions in the jcs Namespace

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

jcs:break-lines( ) Function

SLAX Syntax

```
var $lines = jcs:break-lines(expression);
```

XSLT Syntax

```
<xsl:variable name="lines" select="jcs:break_lines(expression)"/>
```

Description Break a simple element into multiple elements, delimited by newlines. This is especially useful for large output elements such as those returned by the **show pfe** command.

Parameters

- ***expression***—Original output.

Return Value

- **$lines**—Output broken up into lines.

Usage Examples

```
var $lines = jcs:break-lines($output);
for-each ($lines) {
  ...
}
```

jcs:close( ) Function

SLAX Syntax

```
var $results =  jcs:close(connection);
```

XSLT Syntax

```
<xsl:variable name="results" select="jcs:close(connection)"/>
```

Description Close a previously opened connection handle.

Parameters

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

Usage Examples The following example closes the connection handle **$connection**, which was originally generated by a call to the **jcs:open( )** function:

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

jcs:dampen( ) Function

SLAX Syntax

```
var $result = jcs:dampen(tag-string, max, interval);
```

XSLT Syntax

```
<xsl:variable name="result" select="jcs:dampen(tag-string, max, interval)"/>
```

Description Prevent the same operation from being repeatedly executed within a script. The dampen function returns **false** if the number of calls to the **jcs:dampen( )** function exceeds a ***max*** number of calls in the time interval ***interval***. Otherwise the function returns **true**. The

function parameters include an arbitrary string that is used to distinguish different calls to the **jcs:dampen()** function. This tag is stored in the **/var/run** directory on the device.

Parameters

- ***interval***—Time interval, in minutes.
- ***max***—Maximum number of calls to the **jcs:dampen()** function with a given tag allowed before the function returns **false**. This limit is based on the number of calls within a specified time interval.
- ***tag-string***—Arbitrary string used to distinguish different calls to the **jcs:dampen()** function.

Return Value

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

Usage Examples

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

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

jcs:empty() Function

SLAX Syntax

var $result = jcs:empty(*node-set* | *string*);

XSLT Syntax

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

Description

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

Parameters

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

Return Value

- **result**—Boolean value that is **true** if the argument is empty.

Usage Examples

In the following example, if **$set** is empty, the script executes the enclosed code block.

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

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

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

jcs:execute( ) Function

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

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

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

Parameters

- ***connection***—Connection handle generated by a call to the **jcs:open( )** function.
- ***rpc***—Remote procedure call (RPC) to execute.

Return Value

- **result**—Results of the executed RPC, which includes the contents of the **<rpc-reply>** element, but not the **<rpc-reply>** tag itself. This **$result** variable is the same as that produced by the **jcs:invoke( )** function. By default, the results are in XML format equivalent to the output produced with the **| display xml** option in the CLI.

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

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

In XSLT:

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

jcs:first-of( ) Function

SLAX Syntax var $result = jcs:first-of(*object*, "*expression*");

XSLT Syntax <xsl:variable name="result" select="jcs:first-of(*object*, '*expression*')"/>

Description Return the first nonempty (non-null) item in a list. If all objects in the list are empty, the default expression is returned. This function provides the same functionality as an **if / else-if / else** construct but in a much more concise format.

Parameters

- ***expression***—Default value returned if all objects in the list are empty.
- ***object***—List of objects.

Return Value

- **result**—First nonempty (non-null) item in the object list. If all objects in the list are empty, the default expression is returned.

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

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

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

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

The equivalent XSLT code is:

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

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

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

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

jcs:get-hello() Function

SLAX Syntax var $capabilities = jcs:get-hello(*connection*);

XSLT Syntax <xsl:variable name="capabilities" select="jcs:get-hello(*connection*)"/>

Description Return the session ID and the capabilities of the NETCONF server during a NETCONF session.

During session establishment, the NETCONF server and client application each emit a **<hello>** tag element to specify which operations, or *capabilities*, they support from among those defined in the NETCONF specification or published as proprietary extensions. The **<hello>** tag element encloses the **<capabilities>** tag element and the **<session-id>** tag element, which specifies the session ID for this NETCONF session.

Within the **<capabilities>** tag element, a **<capability>** tag element specifies each supported function. Each capability defined in the NETCONF specification is represented by a uniform resource name (URN). Capabilities defined by individual vendors are represented by uniform resource identifiers (URIs), which can be URNs or URLs.

Parameters

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

Return Value

- **capabilities**—XML node-set that specifies which operations, or *capabilities*, the NETCONF server supports. The node-set also includes the session ID.

Usage Examples In the following code snippet, the user, bsmith, establishes a NETCONF session on the default port with the remote device, fivestar, which is running Junos OS. Since the code does not specify a value for the password, the user is prompted for a password during script execution. Once authentication is established, the code calls the **jcs:get-hello()** function and stores the return value in the variable **$hello**, which is then printed to the CLI.

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

The CLI displays the following output:

```
bsmith@fivestar's password:

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

20847
```

jcs:get-input() Function

SLAX Syntax var $user-input = jcs:get-input(*string*);

XSLT Syntax <xsl:variable name="user-input" select="jcs:get-input(*string*)"/>

Description Invoke a CLI prompt and wait for user input. The user input is defined as a string for subsequent use. If the script is run non-interactively, the function returns an empty value. This function cannot be used with event scripts.

Parameters
- ***string***—CLI prompt text.

Return Value
- **user-input**—Text typed by the user and stored as a string. The return value will be empty if the script is run non-interactively.

Usage Examples In the following example, the user is prompted to enter a login name. The user's input is stored in the variable **$username**:

```
var $username = jcs:get-input("Enter login id: ");
```

jcs:get-protocol() Function

SLAX Syntax var $protocol = jcs:get-protocol(*connection*);

XSLT Syntax <xsl:variable name="protocol" select="jcs:get-protocol(*connection*)"/>

Description Return the session protocol associated with the connection handle. The protocol values are **junoscript**, **netconf**, and **junos-netconf**.

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

Return Value
- **protocol**—Session protocol associated with the connection handle. The values are **junoscript**, **netconf**, or **junos-netconf**.

Usage Examples In the following code snippet, the user, bsmith, establishes a NETCONF session on the default port with the remote device, fivestar. Since the code does not specify a value for the password, the user is prompted for a password during script execution. Once authentication is established, the code calls the **jcs:get-protocol()** function and stores the return value in the variable **$protocol**, which is then printed to the CLI.

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

The CLI displays the following output:


```
bsmith@fivestar's password:

netconf
```

jcs:get-secret() Function

SLAX Syntax var $user-input = jcs:get-secret(*string*);

XSLT Syntax <xsl:variable name="user-input" select="jcs:get-secret(*string*)"/>

Description Invoke a CLI prompt and wait for user input. Unlike the **jcs:get-input()** function, the input is not echoed back to the user, which makes the function useful for obtaining passwords. The user input is defined as a string for subsequent use. This function cannot be used with event scripts.

Parameters
- ***string***—CLI prompt text.

Return Value
- **user-input**—Text typed by the user and stored as a string.

Usage Examples The following example shows how to prompt for a password that is not echoed back to the user:

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

jcs:hostname() Function

SLAX Syntax var $name = jcs:hostname(*expression*);

XSLT Syntax <xsl:variable name="name" select="jcs:hostname(*expression*)"/>

Description Return the fully qualified domain name associated with a given IPv4 or IPv6 address. The DNS server must be configured on the device in order to resolve the domain name.

Parameters
- ***expression***—IPv4 or IPv6 address.

Return Value
- **name**—Hostname associated with the IP address.

Usage Examples The following example initializes the variable **address** with the IP address **10.10.10.1**. The **$address** variable is passed as the argument to the **jcs:hostname()** function. If the DNS server is configured on the device, the function will resolve the IP address return the fully qualified domain name, which is stored in the variable **host**.

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

In XSLT:

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

jcs:invoke() Function

SLAX Syntax var $result = jcs:invoke(*rpc*);

XSLT Syntax

```
<xsl:variable name="result" select="jcs:invoke(rpc)"/>
```

Description Invoke a remote procedure call (RPC) on the local device. The function is called with one argument, either a string containing a Junos XML or Junos XML protocol RPC method name or a tree containing an RPC. The result is the contents of the **<rpc-reply>** element, not including the **<rpc-reply>** tag. An RPC allows you to perform functions equivalent to the Junos OS operational mode commands.

Parameters

- *rpc*—String containing a Junos XML or Junos XML protocol RPC method name or a tree containing an RPC.

Return Value

- **result**—Results of the executed RPC, which includes the contents of the **<rpc-reply>** element, but not the **<rpc-reply>** tag itself. By default, the results are in XML format equivalent to the output produced with the **| display xml** option in the CLI.

Usage Examples In the following example, there is a test to see if the **interface** argument is included on the command line when the script is executed. If the argument is provided, the operational mode output of the **show interfaces terse** command is narrowed to include information about that interface only.

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

In this example, the **jcs:invoke( )** function calls an RPC without modifying the output:

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

jcs:open( ) Function

SLAX Syntax

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

var $connection = jcs:open(remote-hostname, <username>, <passphrase>);

var $connection = jcs:open(remote-hostname, <session-options>);
```

XSLT Syntax

```
<xsl:variable name="connection" select="jcs:open()"/>

<xsl:variable name="connection" select="jcs:open(remote-hostname, <username>,
<passphrase>)"/>

<xsl:variable name="connection" select="jcs:open(remote-hostname,
<session-options>)"/>
```

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

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

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

Parameters

- ***remote-hostname***—Domain name or IP address of the remote router, switch, or security device. If you are opening a local connection, do not pass this value. If you specify a session type, this parameter is required.
- ***username***—(Optional) User's login name. If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script.
- ***passphrase***—(Optional) User's login passphrase. If you do not specify a passphrase and it is required for authentication, you should be prompted for one during script execution by the device to which you are connecting.
- ***session-options***—(Optional) XML node-set that specifies the session protocol and connection parameters. The structure of the node-set is:

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

  - **<method>**—(Optional) Session protocol. The protocol is one of three values: **junoscript**, **netconf**, or **junos-netconf**. If you do not specify a protocol, a **junoscript** session is created by default. A **<method>** value of **junoscript** establishes a session with the Junos XML protocol server on a device running Junos OS. A **<method>** value of **netconf** establishes a session with a NETCONF server over an SSHv2 connection. A **<method>** value of **junos-netconf** establishes a session with a NETCONF server over an SSHv1 connection on a device running Junos OS.
  - **<username>**—(Optional) User's login name. If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script.

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

Return Value

- **connection**—Connection handle to the remote host.

Usage Examples

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

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

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

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

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

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

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

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

jcs:output() Function

SLAX Syntax

```
expr jcs:output('string');
```

XSLT Syntax

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

Description

Generate unformatted output text that is immediately sent to the CLI session. In contrast, most script output is output at the end of the script.

Parameters

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

Usage Examples

SLAX syntax:

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

XSLT syntax:

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

jcs:parse-ip( ) Function

SLAX Syntax var $result = jcs:parse-ip("*ipaddress/*(*prefix-length* | *netmask*)");

XSLT Syntax <xsl:variable name="result" select="jcs:parse-ip('*ipaddress/*(*prefix-length* | *netmask*')"/>

Description Parse an IPv4 or IPv6 address.

Parameters

- ***ipaddress***—IPv4 or IPv6 address.
- ***prefix-length***—Prefix length defining the number of bits used in the network prefix portion of the address.
- ***netmask***—Netmask defining the network prefix portion of the address.

Return Value

- **result**—An array containing:
  - Host IP address (or **NULL** in the case of an error)
  - Protocol family (inet for IPv4 or inet6 for IPv6)
  - Prefix length
  - Network address
  - Network mask in dotted decimal notation for IPv4 addresses (left blank for IPv6 addresses)

Usage Examples In the following examples, an IPv4 address and an IPv6 address are parsed and the resulting output is detailed:

```
var $addr = jcs:parse-ip("10.1.2.10/255.255.255.0");
```

- **$addr[1]** contains the host address **10.1.2.10**.
- **$addr[2]** contains the protocol family **inet**.
- **$addr[3]** contains the prefix length **24**.
- **$addr[4]** contains the network address **10.1.2.0**.
- **$addr[5]** contains the netmask for IPv4 **255.255.255.0**.

```
var $addr = jcs:parse-ip("2001:DB8::c50:8a:800:200C:417A/32");
```

- **$addr[1]** contains the host address **2001:db8:0:c50:8a:800:200c:417a**.
- **$addr[2]** contains the protocol family **inet6**.
- **$addr[3]** contains the prefix length **32**.
- **$addr[4]** contains the network address **2001:db8::**.
- **$addr[5]** is blank for IPv6 (**""**).

jcs:printf( ) Function

SLAX Syntax expr jcs:printf(*expression*);

XSLT Syntax <xsl:value-of select="jcs:printf(*expression*)"/>

Description Generate formatted output text. Most standard **printf** formats are supported, in addition to some Junos OS–specific formats. The function returns a formatted string but does not print it on call. To use the following Junos OS modifiers, place the modifier between the percent sign (%) and the conversion specifier.

- **j1**—operator emits the field only if it changed from the last time the function was called. This assumes that the expression's format string is unchanged.
- **jc**—operator capitalizes the first letter of the associated output string.
- **jt{TAG}**—operator emits the tag if the associated argument is not empty.

Parameters

- *expression*—format string containing an arbitrary number of format specifiers and associated arguments to output.

Usage Examples In the following example, the **j1** operator suppresses printing the interface identifier **so-0/0/0** in the second line of output, because the identifier argument has not changed from the first printing. The **jc** operator capitalizes the output strings **up** and **down**. The **jt{--}** operator does not print the **{--}** tag in the first line of output, because the associated output argument is an emtpy string. However, the tag is printed in the second line because the associated output is the non-empty string **test**.

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

produces the following output:

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

jcs:progress( ) Function

SLAX Syntax expr jcs:progress('*string*');

XSLT Syntax <xsl:value-of select="jcs:progress('*string*')"/>

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

Parameters

- ***string***—text output to CLI session

Usage Examples SLAX syntax:

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


XSLT syntax:

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

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

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

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

jcs:regex( ) Function

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

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

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

Parameters

- ***pattern***—Regular expression that is evaluated against the string argument.
- ***string***—String within which to search for matches of the specified regular expression.

Return Value

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

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

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

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

jcs:sleep( ) Function

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

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

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

Parameters

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

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

SLAX syntax:

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

XSLT syntax:

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

jcs:split( ) Function

SLAX Syntax var $substrings = jcs:split(*expression*, *string*, <*limit*>);

XSLT Syntax <xsl:variable name="substrings" select="jcs:split(*expression*, *string*, <*limit*>)"/>

Description Split a string into an array of substrings delimited by a regular expression pattern. If the optional integer argument ***limit*** is specified, the function splits the entire string into ***limit*** number of substrings. If there are more than ***limit*** number of matches, the substrings include the first ***limit***-1 matches as well as the remaining portion of the original string for the last match.

Parameters

- ***expression***—Regular expression pattern used as the delimiter.
- ***limit***—(Optional) Number of substrings into which to break the original string.
- ***string***—Original string.

Return Value

- **$substrings**—Array of ***limit*** number of substrings. If ***limit*** is not specified, the result array size is equal to the number of substrings extracted from the original string as determined by the specified delimiter.

Usage Examples

In the following example, the original string is "123:abc:456:xyz:789". The jcs:split() function breaks this string into substrings that are delimited by the regular expression pattern, which in this case is a colon(:). The optional parameter ***limit*** is not specified, so the function returns an array containing all the substrings that are bounded by the delimiter(:).

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

returns:

```
$substrings[1] == "123"
$substrings[2] == "abc"
$substrings[3] == "456"
$substrings[4] == "xyz"
$substrings[5] == "789"
```

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

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

returns:

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

jcs:sysctl() Function

SLAX Syntax

```
var $value = jcs:sysctl(sysctl-value, "(i | s)");
```

XSLT Syntax

```
<xsl:variable name="value" select="jcs:sysctl(sysctl-value, '(i | s)')"/>
```

Description

Return the given **sysctl** value as a string or an integer. Use the "**i**" argument to specify an integer. Use the "**s**" argument to specify a string.

Parameters

- ***sysctl-value***—**sysctl** value to convert to a string or integer.

Return Value

- **$value**—Returned string or integer value.

Usage Examples

```
var $value = jcs:sysctl("kern.hostname", "s");
```

jcs:syslog() Function

SLAX Syntax

```
expr jcs:syslog(priority, message, <message2>, <message3> ...);
```

XSLT Syntax <xsl:value-of select="jcs:syslog(*priority*, *messsage*, <*message2*>, <*message3*>)"/>

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

Parameters

- ***message***—String or variable that is output to the system log file.
- ***message2***—(Optional) Any additional number of strings or variable names passed as arguments to the function. These are concatenated with the ***message*** argument and output to the system log file.
- ***priority***—Priority given to the syslog message. The priority can be specified as a ***facility.severity*** string, or it can expressed as an integer calculated from the corresponding numeric values of the facility and severity strings. Table 2 on page 30 and Table 3 on page 31 show the facility and severity strings available and their corresponding numeric values.

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

Table 2: Facility Strings

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

Table 3: Severity Strings

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

Usage Examples

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

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

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

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

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

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

jcs:trace( ) Function

SLAX Syntax

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

XSLT Syntax

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

Description

Issue a trace message, which is sent to the trace file. You must configure **traceoptions** under the respective script type in the configuration hierarchy in order to output the **jcs:trace** message to the trace file. The output goes to configured trace file. If **traceoptions** is enabled, but no trace file is explicitly configured, the output goes to the default trace file for that script type.

Parameters

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

Usage Examples

SLAX syntax:

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

XSLT syntax:

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

Junos Script Automation: Named Templates in the jcs Namespace Overview

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

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

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

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

XSLT Syntax

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

SLAX Syntax

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

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

Related Documentation

- Junos Extension Functions in the jcs Namespace Summary on page 12
- Junos Named Templates in the jcs Namespace Summary on page 33
- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 11
- Junos Script Automation: Global Parameters and Variables in the junos.xsl File on page 42

Junos Named Templates in the jcs Namespace Summary

The Junos named templates are summarized in the following table:

Table 4: Junos Named Templates

Template	Description
jcs:edit-path	Generate an **<edit-path>** element suitable for inclusion in an **<xnm:error>** or **<xnm:warning>** element.
jcs:emit-change	Generate a **<change>** or **<transient-change>** element, which results in a persistent or transient change to the configuration.
jcs:emit-comment	Emit a simple comment that indicates a change was made by a commit script.
jcs:grep	Search a file for all instances matching a specified regular expression and write the matching strings and corresponding lines to the result tree.
jcs:load-configuration	Make structured changes to the Junos OS configuration using an op script.
jcs:statement	Generate a **<statement>** element suitable for inclusion in an **<xnm:error>** or **<xnm:warning>** element.

Related Documentation

- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 11
- Junos Script Automation: Named Templates in the jcs Namespace Overview on page 32
- Junos Script Automation: Global Parameters and Variables in the junos.xsl File on page 42

Junos Named Templates in the jcs Namespace

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

jcs:edit-path Template

XSLT Syntax

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

SLAX Syntax

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

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

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

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

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

```
</xsl:if>
```

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

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

jcs:emit-change Template

XSLT Syntax

```
<xsl:call-template name="jcs:emit-change">
  <xsl:with-param name="content">
    ...
  </xsl:with-param>
  <xsl:with-param name="dot" select="expression"/>
  <xsl:with-param name="message">
    <xsl:text>message</xsl:text>
  </xsl:with-param>
  <xsl:with-param name="name" select="name($dot)"/>
  <xsl:with-param name="tag" select="'(change | transient-change)'"/>
</xsl:call-template>
```

SLAX Syntax

```
call jcs:emit-change($dot=expression, $name = name($dot), $tag = "(change |
transient-change)" {
  with $content = {
    ...
  }
  with $message = {
    expr "message";
  }
}
```

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

Parameters This template includes the following optional parameters:

- **content**—Content of the persistent or transient change, relative to **dot**.
- **dot**—XPath expression specifying the hierarchy level at which the change will be made. The default location is the position in the XML hierarchy that the script is currently evaluating. You can alter the default when you call the template by including a valid XPath expression either for the **$dot** parameter in SLAX scripts or for the **select** attribute of the **dot** parameter in XSLT scripts.
- **message**—Warning message displayed in the CLI notifying the user that the configuration has been changed. The message parameter automatically includes the edit path, which defaults to the current location in the XML hierarchy. To change the default edit path, specify a valid XPath expression either for the **$dot** parameter in SLAX scripts or for the **select** attribute of the **dot** parameter in XSLT scripts.

- **name**—Allows you to refer to the current element or attribute. The **name()** XPath function returns the name of an element or attribute. The **name** parameter defaults to the value **name($dot)**, which is the name of the element in **dot** (which in turn defaults to "**.**", which is the current element).
- **tag**—Type of change to generate. By default, the **jcs:emit-change** template generates a persistent change, as designated by the **'change'** expression. To specify a transient change, you must include the **tag** parameter and include the **'transient-change'** expression.

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

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

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

```
[edit]

user@host# commit
[edit interfaces interface so-1/2/3 unit 0]
    warning: Adding 'family mpls' to ISO-enabled interface
[edit interfaces interface so-1/2/3 unit 0]
    warning: Adding ISO-enabled interface so-1/2/3.0 to [protocols mpls]
[edit interfaces interface so-1/3/2 unit 0]
    warning: Adding 'family mpls' to ISO-enabled interface
[edit interfaces interface so-1/3/2 unit 0]
    warning: Adding ISO-enabled interface so-1/3/2.0 to [protocols mpls]
commit complete
```

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

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



```
</xsl:with-param>
```

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

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

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

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

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

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

See also Example: Imposing a Minimum MTU Setting.

jcs:emit-comment Template

XSLT Syntax

```
<junos:comment>
  <xsl:text>...</xsl:text>
</junos:comment>
```

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

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

```
<xsl:call-template name="jcs:emit-change">
  <xsl:with-param name="content">
```

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

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

```
[edit]

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

jcs:grep Template

XSLT Syntax

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

SLAX Syntax

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

Description

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

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

```
<match> {
  <input>
  <output>
}
```

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

Parameters

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

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

- **pattern**—(Mandatory) Regular expression.

Usage Examples Example: Searching Files Using an Op Script

jcs:load-configuration Template

XSLT Syntax

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

SLAX Syntax

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

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

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

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

Parameters

- **action**—Specifies how to load the configuration changes with respect to the candidate configuration. The following options are supported:
  - **merge**—Combine the candidate configuration and the incoming configuration changes. If the candidate configuration and the incoming configuration contain conflicting statements, the incoming statements override those in the candidate configuration.
  - **override**—Replace the entire candidate configuration.
  - **replace**—Replace existing statements in the candidate configuration with the tags of the same name that are marked with **replace:** in the incoming configuration. If

there is no existing statement of the same name in the candidate configuration, the statement is added to the candidate configuration.

- **commit-options**—Node-set defining options that customize the commit command. The default value is null. Supported commit options are:
  - **check**—Check the correctness of the candidate configuration syntax, but do not commit the changes.
  - **force-synchronize**—Force the commit on the other Routing Engine (ignore any warnings).
  - **log**—Write the specified message to the commit log. This is identical to the CLI configuration mode command **commit comment**.
  - **synchronize**—Synchronize the commit on both Routing Engines.
- **configuration**—XML configuration changes.
- **connection**—Connection handle generated by a call to the **jcs:open()** function.

Usage Examples

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

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

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

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

SLAX syntax:

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

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

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

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

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

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

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

jcs:statement Template

XSLT Syntax

```
<xsl:call-template name="jcs:statement">
  <xsl:with-param name="dot" select="expression"/>
</xsl:call-template>
```

SLAX Syntax

```
call jcs:statement($dot=expression);
```

Description Generate a **<statement>** element suitable for inclusion in an **<xnm:error>** or **<xnm:warning>** element. This location defaults to " **.** ", the current position in the XML hierarchy. If the error is not at the current position in the XML hierarchy, you can alter the default when

you call the template by including a valid XPath expression either for the **$dot** parameter in SLAX scripts or for the **select** attribute of the **dot** parameter in XSLT scripts.

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

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

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

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

```
[edit]

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

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

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

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

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

XSLT Syntax

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0">
```



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

SLAX Syntax

```
version 1.0;
import "../import/junos.xsl";
```

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

- Global Parameters on page 43
- Global Variable on page 44

Global Parameters

Several predefined global parameters are available for use in commit, op, and event scripts. The parameters provide information about the Junos OS environment. Table 5 on page 43 describes the built-in arguments.

Table 5: Predefined Parameters Available to Automation Scripts

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

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

SLAX syntax:

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

XSLT syntax:

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

Global Variable

Starting with Junos OS Release 11.1, Junos OS also provides a single global variable, **$junos-context**, which is accessible for use in all commit, op, or event scripts that import the **junos.xsl** file. The **$junos-context** variable is a node-set, which has elements that mirror the original global parameters described in "Global Parameters" on page 43 as well as additional elements with information about the Junos OS environment, such as whether a script is executed on the master Routing Engine.

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

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

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

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

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

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

Table 6 on page 45 identifies each node of the **$junos-context** variable node-set, provides a brief description of the node, and gives examples of values for any elements that are not input to a script as an empty tag.

Table 6: Global Variable $junos-context Available to Automation Scripts

Parent Node	Node	Description	Example content
<junos-context>	<chassis>	Specifies whether the script is executed on a component of a routing matrix, the Root System Domain (RSD), or a Protected System Domain (PSD)	scc, lcc (TX Matrix) psd, rsd (JCS) others
	<hostname>	Hostname of the local device	Tokyo
	<localtime>	Local time when the script is executed	Fri Dec 10 11:42:21 2010
	<localtime-iso>	Local time, in ISO format, when the script is executed	2010-12-10 11:42:21 PST
	<pid>	cscript process ID	5257
	<product>	Model of the local device	m10i
	<re-master/>	Empty element included if the script is executed on the master Routing Engine	
	<routing-engine-name>	Routing Engine on which the script is executed	re0
	<tty>	TTY of the user's session	/dev/ttyp1
	<script-type>	Type of script being executed	op
<junos-context> <user-context>	<class-name>	Login class of the user executing the script	superuser
	<login-name>	Login name of the user executing the script. For AAA access, this is the RADIUS/TACACS username.	jsmith
	<uid>	User ID number of the user executing the script as defined in the device configuration	2999
	<user>	Local name of the user executing the script. Junos OS uses the local name for authentication. It might differ from the **login-name** used for AAA authentication.	root
<junos-context> <op-context> (op scripts only)	<via-url>	Empty element included if the remote op script is executed using the **op url** command	

Table 6: Global Variable $junos-context Available to Automation Scripts *(continued)*

Parent Node	Node	Description	Example content
<junos-context> <commit-context> (commit scripts only)	<commit-boot/>	Empty element included when the commit occurs at boot time	
	<commit-check/>	Empty element included when a **commit check** is performed	
	<commit-comment>	User comment regarding the commit	Commit to fix forwarding issue
	<commit-confirm/>	Empty element included when a **commit confirmed** is performed	
	<commit-sync/>	Empty element included when a **commit synchronize** is performed	

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

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

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

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

Related Documentation

- Junos Script Automation: Extension Functions in the jcs Namespace Overview on page 11
- Junos Script Automation: Named Templates in the jcs Namespace Overview on page 32
- SLAX Parameters Overview on page 81
- SLAX Variables Overview on page 84
- XSLT Parameters Overview on page 60
- XSLT Variables Overview on page 63

CHAPTER 4

XML

XML Overview

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

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

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

The following sections discuss general aspects of XML:

Tag Elements

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

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

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

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

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

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

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

Attributes

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

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

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

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

Namespaces

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

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

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

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

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

Document Type Definition

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

Related Documentation

- Junos XML API and Junos XML Management Protocol Overview on page 7
- XML and Junos OS on page 49

XML and Junos OS

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

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

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

You can view the XML-formatted output of any operational mode command by issuing the command in the CLI and adding the **| display xml** option. The following example shows the text-fomatted and XML-formatted output for the **show chassis alarms** operational mode command:

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

user@host> show chassis alarms | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4B1/junos">
    <alarm-information xmlns="http://xml.juniper.net/junos/10.4B1/junos-alarm">
        <alarm-summary>
            <no-active-alarms/>
        </alarm-summary>
    </alarm-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

You can view the Junos XML API representation of any operational mode command by issuing the command in the CLI and adding the **| display xml rpc** option. The following example shows the Junos XML API tag element for the **show chassis alarms** command.

```
user@host> show chassis alarms | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4B1/junos">
    <rpc>
        <get-alarm-information>
        </get-alarm-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

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

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

Junos automation scripts use XML to communicate with the host device. Junos OS provides XML-formatted input to a script. The script processes the input source tree and

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

Related Documentation

- *Junos XML API Configuration Reference*
- *Junos XML API Operational Reference*

CHAPTER 5

XSLT

XSLT Overview

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

XSLT Advantages

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

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

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

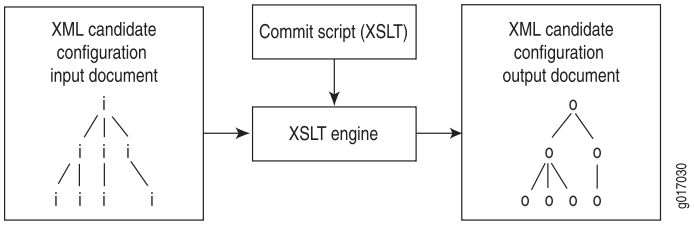
Although XSLT provides a powerful scripting ability, its focus is specific and limited. It does not make Junos OS vulnerable to arbitrary or malicious programmers. XSLT restricts programmers from performing haphazard operations, such as opening random Transmission Control Protocol (TCP) ports, forking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

XSLT Engine

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

Figure 1 on page 54 shows the relationship between an XSLT commit script and the XSLT engine.

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



XSLT Concepts

XSLT has seven basic concepts. These are summarized in Table 7 on page 54.

Table 7: XSLT Concepts

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

Table 7: XSLT Concepts *(continued)*

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

Related Documentation

- XPath Overview on page 55
- XSLT Context (Dot) Overview on page 68
- XSLT Parameters Overview on page 60
- XSLT Programming Instructions Overview on page 64
- XSLT Recursion Overview on page 67
- XSLT Templates Overview on page 58
- XSLT Variables Overview on page 63

XSLT Namespace

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

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

Once the XSLT namespace is declared in a script, you use elements and attributes from the namespace by adding the associated prefix, which in this case is **xsl**, to the tag or attribute name. In the preceding example, the XSLT processor knows to treat **xsl:template** as an XSLT instruction. During processing, the **xsl** prefix is expanded into the URI reference, and the functionality of the **template** element is defined by the XSLT namespace. For more information about namespaces, see "XML Overview" on page 47.

XPath Overview

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

Nodes and Axes

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

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

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

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

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

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

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

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

Path and Predicate Syntax

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

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

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

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

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

XPath Operators

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

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

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

XSLT Templates Overview

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

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

- Unnamed (Match) Templates on page 58
- Named Templates on page 59

Unnamed (Match) Templates

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

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

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

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

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

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

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

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

Named Templates

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

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

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

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

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

When you invoke a named template, you can pass arguments into the template by including the **<xsl:with-param>** child element and specifying the **name** attribute. The value of the **<xsl:with-param> name** attribute must match a parameter defined in the actual template; otherwise the parameter is ignored. Optionally, you can set a value for each parameter with either the **select** attribute or the content of the **<xsl:with-param>** element. If you do not define a value for the parameter in the calling environment, the script passes in the current value of the parameter if it was previously initialized, or it generates an error if the parameter was never declared. For more information about passing parameters, see "XSLT Parameters Overview" on page 60.

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

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

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

Related Documentation

- XSLT Parameters Overview on page 60
- xsl:apply-templates on page 146
- xsl:call-template on page 146
- xsl:param on page 152
- xsl:template on page 154
- xsl:template match="/" Template
- xsl:with-param on page 157

XSLT Parameters Overview

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

Declaring Parameters

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

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

<!-- global parameter -->
<xsl:param name="interface1"/>
```

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

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

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

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

Passing Parameters

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

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

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

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

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

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

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

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

Example: Parameters and Match Templates

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

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

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

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

Example: Parameters and Named Templates

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

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

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

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

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

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



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

Related Documentation

- XSLT Templates Overview on page 58
- xsl:param on page 152
- xsl:with-param on page 157

XSLT Variables Overview

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

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

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

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

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

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

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

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

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

Table 8 on page 64 provides examples of XSLT variable declarations along with pseudocode explanations.

Table 8: Examples and Pseudocode for XSLT Variable Declaration

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

Related Documentation

- xsl:variable on page 156

XSLT Programming Instructions Overview

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

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

- <xsl:choose> Programming Instruction on page 64
- <xsl:for-each> Programming Instruction on page 65
- <xsl:if> Programming Instruction on page 65
- Sample XSLT Programming Instructions and Pseudocode on page 66

<xsl:choose> Programming Instruction

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

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

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

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

<xsl:for-each> Programming Instruction

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

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

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

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

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

<xsl:if> Programming Instruction

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

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

There is no corresponding else clause.

Sample XSLT Programming Instructions and Pseudocode

Table 9 on page 66 presents examples that use several XSLT programming instructions along with pseudocode explanations.

Table 9: Examples and Pseudocode for XSLT Programming Instructions

Programming Instruction	Pseudocode Explanation
<xsl:choose> <xsl:when test="system/host-name"> <change> <system> <host-name>M320</host-name> </system> </change> </xsl:when> <xsl:otherwise> <xnm:error> <message> Missing [edit system host-name] M320. </message> </xnm:error> </xsl:otherwise> </xsl:choose>	When the **host-name** statement is included at the **[edit system]** hierarchy level, change the hostname to **M320**. Otherwise, issue the warning message: **Missing [edit system host-name] M320**.
<xsl:for-each select="interfaces/interface[starts-with(name, 'ge-')]/unit">	For each Gigabit Ethernet interface configured at the **[edit interfaces ge-*fpc/pic/port* unit *logical-unit-number*]** hierarchy level.

<xsl:for-each select="data[not(value)]/name">	Select any macro parameter that does not contain a parameter value.
	
	In other words, match all **apply-macro** statements of the following form:
	
	apply-macro *apply-macro-name* {
	parameter-name;
	}
	
	And ignore all **apply-macro** statements of the form:
	
	apply-macro *apply-macro-name* {
	parameter-name parameter-value;
	}

<xsl:if test="not(system/host-name)">	If the **host-name** statement is not included at the **[edit system]** hierarchy level.
<xsl:if test="apply-macro[name = 'no-igp']	If the **apply-macro** statement named **no-igp** is included at the current hierarchy level.
<xsl:if test="not(../apply-macro[name = 'no-ldp'])	If the **apply-macro** statement with the name **no-ldp** is not included two hierarchy levels above the current hierarchy level.

Related Documentation

- xsl:choose on page 147

- xsl:for-each on page 149
- xsl:if on page 150
- xsl:otherwise on page 151
- xsl:when on page 157

XSLT Recursion Overview

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

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

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

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

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

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

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

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

XSLT Context (Dot) Overview

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

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

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

CHAPTER 6

SLAX

SLAX Overview

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

SLAX Advantages

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

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

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

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

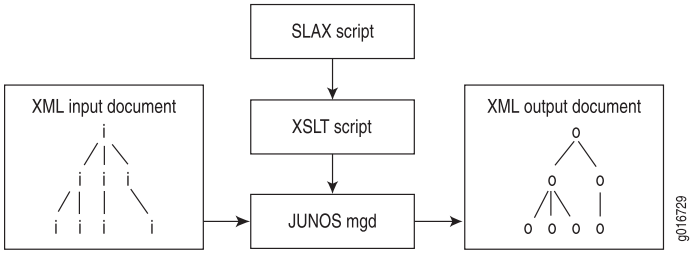
How SLAX Works

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

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

Figure 2 on page 70 shows the flow of SLAX script input and output.

Figure 2: SLAX Script Input and Output



Related Documentation

- Converting Scripts Between SLAX and XSLT on page 71
- SLAX Elements and Element Attributes Overview on page 75

- SLAX Statements Overview on page 85
- SLAX Syntax Rules Overview on page 73
- SLAX Templates Overview on page 77
- SLAX Variables Overview on page 84
- XPath Expressions Overview for SLAX on page 76
- XSLT Overview on page 53

Converting Scripts Between SLAX and XSLT

SLAX is a C-like alternative syntax to XSLT and can be viewed as a preprocessor for XSLT. Before Junos OS invokes the XSLT processor, the software converts any SLAX constructs in the script (such as **if/then/else**) to equivalent XSLT constructs (such as **<xsl:choose>** and **<xsl:if>**). For more information about SLAX, see "SLAX Overview" on page 69.

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

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

- Converting a Script from SLAX to XSLT on page 71
- Converting a Script from XSLT to SLAX on page 72

Converting a Script from SLAX to XSLT

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

The command syntax is:

user@host> **request system scripts convert slax-to-xslt source *source* destination *dest***

The following three examples show the command using a source and destination directory relevant to the default storage location for the type of script being converted:

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

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

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

```
conversion complete
```

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

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

If you specify only the destination directory and do not specify a destination filename, the generated file is named **SLAX-Conversion-Temp.*xxxxx*** where ***xxxxx*** is a randomly generated series of characters.

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

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

Converting a Script from XSLT to SLAX

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

The command syntax is:

user@host> **request system scripts convert xslt-to-slax source *source* destination *dest***

The following three examples show the command using a source and destination directory relevant to the default storage location for the type of script being converted:

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

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

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

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

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

If you specify only the destination directory and do not specify a destination filename, the generated file is named **SLAX-Conversion-Temp.*xxxxx*** where ***xxxxx*** is a randomly generated series of characters.

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

user@host> file list /var/db/scripts/op
SLAX-Conversion-Temp.Vosnd
script1.xsl
```

Related Documentation

- SLAX Overview on page 69

SLAX Syntax Rules Overview

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

- Code Blocks on page 73
- Comments on page 73
- Line Termination on page 74
- Strings on page 74

Code Blocks

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

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

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

Comments

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

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

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

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

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

The XSLT equivalent is:

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

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

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

Line Termination

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

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

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

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

Strings

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

For example:

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

Related Documentation

- SLAX Elements and Element Attributes Overview on page 75
- SLAX Overview on page 69
- SLAX Statements Overview on page 85SLAX Templates Overview on page 77
- SLAX Templates Overview on page 77
- SLAX Variables Overview on page 84

SLAX Elements and Element Attributes Overview

SLAX Elements

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

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

The XSLT equivalent is:

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

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

SLAX Element Attributes

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

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

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

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

The XSLT equivalent is:

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

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

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

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

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

The XSLT equivalent is:

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

Related Documentation

- XML Overview on page 47

XPath Expressions Overview for SLAX

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

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

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

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

The XSLT equivalent is:

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

Related Documentation

- concat() on page 139
- SLAX Elements and Element Attributes Overview on page 75
- SLAX Syntax Rules Overview on page 73
- XPath Overview on page 55
- xsl:text on page 155
- xsl:value-of on page 155

SLAX Templates Overview

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

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

- Unnamed (Match) Templates on page 78
- Named Templates on page 79

Unnamed (Match) Templates

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

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

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

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

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

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

The XSLT equivalent:

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

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

Named Templates

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

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

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

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

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

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

In SLAX, you invoke named templates using the **call** statement, which consists of the **call** keyword and template name followed by a set of parameter bindings. These bindings are a comma-separated list of parameter names that are passed into the template from the calling environment. Parameter assignments are made by name and not by position in the list. Alternatively, you can declare parameters inside the **call** block using the **with** statement. Parameters passed into a template must match a parameter defined in the actual template; otherwise the parameter is ignored. Optionally, you can set a value for each parameter. If you do not define a value for the parameter in the calling environment, the script passes in the current value of the parameter if it was previously initialized, or it generates an error if the parameter was never declared. For more information about passing parameters, see "SLAX Parameters Overview" on page 81.

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

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

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

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

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

The XSLT equivalent is:

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

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

Related Documentation

- SLAX Parameters Overview on page 81
- XSLT Templates Overview on page 58
- apply-templates on page 124
- call on page 125
- match on page 130

- param on page 132
- with on page 137

SLAX Parameters Overview

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

Declaring Parameters

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

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

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

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

/* global parameter */
param $interface1 = “fxp0”;
```

Local parameters must be declared at the beginning of a block and their scope is limited to the block in which they are declared. In a template, you declare parameters either in a parameter list or by using the **param** statement in the template block. Optionally, declare default values for each template parameter. If a template is invoked without the parameter, the default expression is evaluated, and the results are assigned to the parameter. If you do not define a default value in the template, the parameter defaults to an empty string.

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

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

An alternative, but equivalent, declaration is:

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

```
  }
}
```

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

Passing Parameters

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

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

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

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

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

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

```
match configuration {
   call print-host-name($message);
   /* If $message was initialized previously, the current value is used;
```

```
    * If $message was declared but not initialized, an empty string is used;
    * If $message was never declared, the call generates an error. */
}
```

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

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

Example: Parameters and Match Templates

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

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

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

The XSLT equivalent is:

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

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

Related Documentation

- SLAX Templates Overview on page 77
- param on page 132
- template on page 134

- with on page 137

SLAX Variables Overview

In SLAX, you declare variables using the **var** statement. In the declaration, the variable name is prefixed with the dollar sign ($), unlike the XSLT declaration, where the dollar sign does not prefix the value of the **name** attribute of the **<xsl:variable>** element. Once you declare a variable, you can reference it within an XPath expression using the variable name prefixed with a dollar sign ($).

Variables are immutable; you can set the value of a variable only when you declare the variable, after which point, the value is fixed. You initialize a variable by following the variable name with an equal sign (=) and an expression. The following example declares and initializes the variable **location**. The **location** variable is then used to initialize the **message** variable.

```
var $location = $dot/@location;
var $message = "We are in " _ $location _ " now.";
```

The XSLT equivalent is:

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

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

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

Variables are immutable. As such, you can never change the value of a variable after it is defined in the declaration. Although you cannot directly update the value of a variable, you can mimic the effect by recursively calling a function and passing in the value of the variable as a parameter. For example:

```
var $count = 1;
match / {
  call update-count($myparam = $count);
}
template update-count($myparam) {
  expr $count _ ", " $myparam _"\n";
  if ($myparam != 4) {
    call update-count($myparam = $myparam + 1)
  }
}
```

Executing the op script in the CLI produces the following output in the log file. Although the **count** variable must remain fixed, **myparam** is updated with each call to the template.

```
1, 1
1, 2
```

```
1, 3
1, 4
1, 5
```

Related Documentation

- XSLT Variables Overview on page 63
- SLAX Parameters Overview on page 81
- **var on page 135**

SLAX Statements Overview

This section lists some commonly used SLAX statements, with brief examples and XSLT equivalents.

- for-each Statement on page 85
- if, else if, and else Statements on page 86
- match Statement on page 87
- ns Statement on page 87
- version Statement on page 88

for-each Statement

The SLAX **for-each** statement functions like the **<xsl:for-each>** element. The statement consists of the **for-each** keyword, a parentheses-delimited expression, and a curly braces-delimited block. The **for-each** statement tells the processor to gather together a set of nodes and process them one by one. The nodes are selected by the specified XPath expression. Each of the nodes is then processed according to the instructions held in the **for-each** code block.

```
for-each (xpath-expression) {
  ...
}
```

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

In the following example, the **inventory** variable stores the inventory hierarchy. The **for-each** statement recursively processes each **chassis-sub-module** node that is a child of **chassis-module** that is a child of the **chassis** node. For each **chassis-sub-module** element that contains a **part-number** with a value equal to the specified part number, a **message** element is created that includes the name of the chassis module and the name and description of the chassis sub module.

```
for-each ($inventory/chassis/chassis-module/
        chassis-sub-module[part-number == '750-000610']) {
  <message> "Down rev PIC in " _ ../name _ ", " _ name _ ": " _ description;
}
```

The XSLT equivalent is:

```
<xsl:for-each select="$inventory/chassis/chassis-module/
        chassis-sub-module[part-number = '750-000610']">
```

```
  <message>
    <xsl:value-of select="concat('Down rev PIC in ', ../name, ', ', name, ': ',
        description)"/>
  </message>
</xsl:for-each>
```

if, else if, and else Statements

SLAX supports **if**, **else if**, and **else** statements. The **if** statement is a conditional construct that causes instructions to be processed if the specified XPath expression evaluates to true. The **if** construct may have one or more associated **else if** clauses, each of which tests an XPath expression. If the expression in the **if** statement evaluates to false, the processor checks each **else if** expression. If a statement evaluates to true, the script executes the instructions in the associated block and ignores all subsequent **else if** and **else** statements. The optional **else** clause is the default code that is executed in the event that all associated **if** and **else-if** expressions evaluate to false. If all of the **if** and **else if** statement evaluate to false, and the **else** statement is not present, no action is taken.

The expressions that appear in parentheses are extended XPath expressions, which support the double equal sign (**==**) in place of XPath's single equal sign (**=**).

```
if (expression) {
  /* If block Statement */
}
else if (expression) {
   /* else if block statement */
}
else {
  /* else block statement */
}
```

During script processing, an **if** statement that does not have an associated **else if** or **else** statement is transformed into an **<xsl:if>** element. If either the **else if** or **else** clauses are present, the **if** statement and associated **else if** and **else** blocks are transformed into an **<xsl:choose>** element.

```
if (starts-with(name, "fe-")) {
  if (mtu < 1500) {
    /* Select Fast Ethernet interfaces with low MTUs */
  }
}
else {
  if (mtu > 8096) {
    /* Select non-Fast Ethernet interfaces with high MTUs */
  }
}
```

The XSLT equivalent is:

```
<xsl:choose>
  <xsl:when test="starts-with(name, 'fe-')">
    <xsl:if test="mtu &lt; 1500">
      <!-- Select Fast Ethernet interfaces with low MTUs -->
    </xsl:if>
  </xsl:when>
  <xsl:otherwise>
    <xsl:if test="mtu &gt; 8096">
```

```
        <!-- Select non-Fast Ethernet interfaces with high MTUs -->
      </xsl:if>
    </xsl:otherwise>
  </xsl:choose>
```

match Statement

You specify basic match templates using the **match** statement, followed by an expression specifying when the template should be allowed and a block of statements enclosed in a set of braces.

```
match configuration {
   <xnm:error> {
     <message> "...";
  }
}
```

The XSLT equivalent is:

```
<xsl:template match="configuration">
  <xnm:error>
    <message> ...</message>
  </xnm:error>
</xsl:template>
```

For more information about constructing match templates, see "SLAX Templates Overview" on page 77.

ns Statement

You specify namespace definitions using the SLAX **ns** statement. This consists of the **ns** keyword, a prefix string, an equal sign, and a namespace Uniform Resource Identifier (URI). To define the default namespace, use only the **ns** keyword and a namespace URI.

```
ns junos = "http://www.juniper.net/junos/";
```

The **ns** statement can appear after the **version** statement at the beginning of the style sheet or at the beginning of any block.

```
ns a = "http://example.com/1";
ns "http://example.com/global";
ns b = "http://example.com/2";
match / {
  ns c = "http://example.com/3";
  <top> {
    ns a = "http://example.com/4";
    apply-templates commit-script-input/configuration;
  }
}
```

When it appears at the beginning of the style sheet, the **ns** statement can include either the **exclude** or **extension** keyword. The keyword instructs the parser to add the namespace prefix to the **exclude-result-prefixes** or **extension-element-prefixes** attribute.

```
ns exclude foo = "http://example.com/foo";
ns extension jcs = "http://xml.juniper.net/jcs";
```

The XSLT equivalent is:

```
<xsl:stylesheet xmlns:foo="http://example.com/foo"
  xmlns:jcs="http://xml.juniper.net/jcs"
  exclude-result-prefixes="foo"
  extension-element-prefixes="jcs">
  <!-- ... -->
</xsl:stylesheet>
```

version Statement

All SLAX style sheets must begin with a **version** statement, which specifies the version number for the SLAX language. The current version is **1.0**. SLAX version 1.0 uses XML version 1.0 and XSLT version 1.1.

```
version 1.0;
```

The XSLT equivalent is:

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

Related Documentation

- else on page 126
- for-each on page 128
- if on page 129
- match on page 130
- version on page 136

XSLT Elements Without SLAX Equivalents

Some XSLT elements are not directly translated into SLAX statements. Some examples of XSLT elements for which there are no SLAX equivalents are **<xsl:fallback>**, **<xsl:output>**, and **<xsl:sort>**.

You can encode these elements directly as normal SLAX elements in the XSLT namespace. For example, you can include the **<xsl:output>** and **<xsl:sort>** elements in a SLAX script, as shown here:

```
<xsl:output method="xml" indent="yes" media-type="image/svg">;
match * {
  for-each (configuration/interfaces/unit) {
    <xsl:sort order="ascending">;
  }
}
```

When you include XSLT namespace elements in a SLAX script, do not include closing tags. For empty tags, do not include a forward slash (/) after the tag name. The examples shown in this section demonstrate the correct syntax.

The following XSLT snippet contains a combination of elements, some of which have SLAX counterparts and some of which do not:

```
<xsl:loop select="title">
  <xsl:fallback>
```

```
      <xsl:for-each select="title">
        <xsl:value-of select="."/>
      </xsl:for-each>
    </xsl:fallback>
  </xsl:loop>
```

The SLAX conversion uses the XSLT namespace for XSLT elements that do not have SLAX counterparts:

```
<xsl:loop select = "title"> {
  <xsl:fallback> {
    for-each (title) {
      expr .;
    }
  }
}
```

SLAX Operators

SLAX provides a variety of operators, which add great versatility to the SLAX scripting language. Table 10 on page 89 summarizes the available operators and provides an example and an explanation of each.

Table 10: SLAX Operators

Name	Operator	Example / Explanation
Addition	+	var $example = 1 + 1; Assigns the value of 1 + 1 to the **$example** variable.
Subtraction, Negation	-	var $example = 1 - 1; Assigns the value of 1 - 1 to the **$example** variable; when used as an unary operator, it changes the sign of a number from positive to negative or from negative to positive.
Multiplication	*	<output>5 * 10; Results in the value 50 being written to the console.
Division	div	<output>$bit-count div 8; Divides the bits by eight, returning the byte count, and displays the result on the console (requires that **$bit-count** has been initialized).
Modulo	mod	<output>10 mod 3; Returns the division remainder of two numbers. In this example, the expression writes 1 to the console.
Equals	==	$mtu == 1500 If $mtu is 1500 then the expression resolves to true, otherwise it returns false (requires that **$mtu** has been initialized).

Table 10: SLAX Operators *(continued)*

Name	Operator	Example / Explanation
Does not equal	!=	$mtu != 1500 If **$mtu** equals 1500 then the result is false, otherwise it returns true (requires that **$mtu** has been initialized)
Less than	<	$hop-count < 15 Returns true if the value to the left of the operator is less than the value to the right, otherwise it returns false. In this example, if $hop-count is less than 15, the expression returns true (requires that **$hop-count** has been initialized).
Less than or equal to	<=	$hop-count <= 14 Returns true if the value to the left of the operator is either less than the value to the right or equal to it, otherwise it returns false. In this example, if **$hop-count** is 14 or less, the expression returns true (requires that **$hop-count** has been initialized).
Greater than	>	$hop-count > 0 Returns true if the value to the left of the operator is greater than the value to the right, otherwise it returns false. In this example, if **$hop-count** is greater than zero, the expression returns true (requires that **$hop-count** has been initialized).
Greater than or equal to	>=	$hop-count >= 1 Returns true if the value to the left of the operator is either greater than the value to the right or equal to it, otherwise it returns false. In this example, if **$hop-count** is one or greater, the expression returns true (requires that **$hop-count** has been initialized).
Parentheses	( )	var $result = ($byte-count * 8) + 150; Used to create complex expressions. Parentheses function the same way as in a mathematical expression, with the expression within the parentheses evaluated first. Parentheses can be nested; the innermost set of parentheses is evaluated first, then the next set, and so on.
And	&&	$byte-count > 500000 && $byte-count < 1000000 Evaluates two expressions and returns one boolean result. If either of the two expressions evaluates to false, then the combined expression evaluates to false.
Or	||	$mtu-size != 1500 || $mtu-size > 2000 Evaluates two expressions and returns one boolean result. If either of the two expressions evaluates to true, then the combined expression evaluates to true.

Table 10: SLAX Operators *(continued)*

Name	Operator	Example / Explanation
String concatenation	_ (under-score)	var $combined-string = $host-name _ " is located at " _ $location; Concatenates multiple strings (note that strings cannot be combined using the + operator in SLAX). In the example, if **$host-name** is "r1" and **$location** is "HQ" then the value of **$combined-string** is "r1 is located at HQ".
Node-Set Union	|	var $all-interface-nodes = $fe-interface-nodes | $ge-interface-nodes; Creates a union of two node-sets. All the nodes from one set combine with the nodes in the second set. This is useful when a script needs to perform a similar operation over XML nodes that are pulled from multiple sources.
Result Tree Fragment to Node-Set Conversion	:=	var $new-node-set := $rtf-variable; A result tree fragment contains an unparsed XML data structure. It is not possible to retrieve any of the embedded XML information from this data type. The **:=** operator converts a variable from a result tree fragment into a node-set. The script can then tell Junos OS to search the node-set for the appropriate information and extract it. Only Junos OS Release 9.2 and beyond support this operator.

Related Documentation

- SLAX Elements and Element Attributes Overview on page 75
- SLAX Overview on page 69
- SLAX Statements Overview on page 85
- SLAX Syntax Rules Overview on page 73
- SLAX Variables Overview on page 84

PART 2

Configuration

CHAPTER 7

Storing and Enabling Scripts

Storing and Enabling Scripts

To use a script on a switch, router, or security device, you must copy the script to the device and enable it in the configuration.

1. Create the script.
2. Copy the script to the appropriate directory on the device for that script type. Only users who belong to the Junos OS **super-user** login class can access and edit files in the script directories on a device running Junos OS.

   By default, scripts are stored in and executed from the **/var/db/scripts** directory on the device's hard drive under the subdirectory appropriate to the script type. EX Series switches use the default directory **/config/db/scripts**. You can also store scripts on the flash drive under the **/config/scripts** directory.

   commit script—Copy the script to the **/var/db/scripts/commit** directory on the hard drive or the **/config/scripts/commit** directory on the flash drive.

   op script—Copy the script to the **/var/db/scripts/op** directory on the hard drive or the **/config/scripts/op** directory on the flash drive.

   event script—Copy the script to the **/var/db/scripts/event** directory on the hard drive or the **/config/scripts/event** directory on the flash drive.

NOTE: If the device has dual Routing Engines and you want to enable the script to execute on both Routing Engines, you must copy it to the appropriate directory on both Routing Engines. The commit synchronize command does not automatically copy scripts between Routing Engines.

3. Enable the script by including the **file *filename*** statement at the appropriate hierarchy level for that script type.

   commit script—Include the **file *filename*** statement at the **[edit system scripts commit]** hierarchy level. For instructions, see Controlling Execution of Commit Scripts During Commit Operations.

   op script—Include the **file *filename*** statement at the **[edit system scripts op]** hierarchy level. For instructions, see Enabling an Op Script and Defining a Script Alias.

   event script—Include the **file *filename*** statement at the **[edit event-options event-script]** hierarchy level. For instructions, see Enabling an Event Script.

4. If you store scripts in and load them from flash memory, include the **load-scripts-from-flash** statement at the **[edit system scripts]** hierarchy level. For detailed information about storing scripts in flash memory, see "Storing Scripts in Flash Memory" on page 96.

   ```
   [edit]
   user@host# set system scripts load-scripts-from-flash
   ```

5. Issue the **commit** command.

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

Newly enabled commit scripts do not execute during the commit operation, but execute automatically during each subsequent commit operation. After the commit operation completes, an op script can be executed on the device. See Executing an Op Script. After the commit operation completes, the event script is loaded into memory and ready for automatic execution in response to system log events. For more information, see Executing Event Scripts in an Event Policy.

Related Documentation

- Storing Scripts in Flash Memory on page 96
- Controlling Execution of Commit Scripts During Commit Operations

Storing Scripts in Flash Memory

By default, scripts are stored in and executed from the **/var/db/scripts/** directory on the device's hard drive under the subdirectory appropriate to the script type.

NOTE: EX Series switches use the default directory **/config/db/scripts/**.

To store scripts in and load them from flash memory instead, include the **load-scripts-from-flash** statement at the **[edit system scripts]** hierarchy level:

```
[edit]
user@host# set system scripts load-scripts-from-flash
```

When you add the **load-scripts-from-flash** statement in the configuration, all commit, event, operation, and script library scripts are loaded from the **/config/scripts/** directory

on the flash drive under the subdirectory appropriate to the script type. You must manually move scripts from the hard drive to the flash drive. They are not moved automatically. Similarly, if you delete the **load-scripts-from-flash** statement from the configuration, you must manually copy the scripts from the flash drive to the hard drive to ensure that the current versions of the scripts are executed. Changing the scripts' physical location has no effect on their operation.

The **/var/run/scripts/** directory always links to the directory from which the scripts are loaded. If you do not set the **load-scripts-from-flash** statement in the configuration, **/var/run/scripts/** points to the **/var/db/scripts/** directory on the device's hard drive. If you set the **load-scripts-from-flash** statement in the configuration, **/var/run/scripts/** points to the **/config/scripts/** directory in flash memory.

To view the scripts currently on the device, list the contents of **/var/run/scripts/*type*/**, where *type* is the subdirectory appropriate to the script type. In the following example, the **load-scripts-from-flash** statement is not configured. In this case, **/var/run/scripts/commit/** points to the**/var/db/scripts/commit/** directory on the hard drive. Listing the files for **/var/run/scripts/commit/** is identical to listing the files in the **/var/db/scripts/commit/** directory.

```
user@host>file list /var/run/scripts/commit

/var/run/scripts/commit:
commit-changes-load-replace.slax
commit-protect.slax

user@host> file list /var/db/scripts/commit

/var/db/scripts/commit:
commit-changes-load-replace.slax
commit-protect.slax

user@host> file list /config/scripts/commit

/config/scripts/commit:
```

Storing Scripts and Script Functionality in the Script Library

Starting with Junos OS Release 11.1, Junos OS provides a dedicated directory for script libraries, where users can store scripts or script functionality that then can be imported into any commit, event, or op script. Upon installation, Junos OS creates the **/var/db/scripts/lib/** directory with permissions identical to the **commit/**, **event/**, and **op/** directories at the same hierarchy level.

NOTE: On EX Series switches, Junos OS creates the **/config/db/scripts/lib/** directory.

Junos OS will not overwrite or erase any files in an existing **lib/** directory upon installation or upgrade.

If you configure the **load-scripts-from-flash** option at the **[edit system scripts]** hierarchy level, Junos OS creates the **/config/scripts/lib/** directory. When you add or remove the **load-scripts-from-flash** statement in the configuration, you must manually move scripts and script libraries from the hard drive to the flash drive, or vice versa, as appropriate. They are not moved automatically.

To import a script from the script library, include the **<xsl:import>** tag in the style sheet declaration of an XSLT script or include the **import** statement in a SLAX script and specify the file location as shown in the following sample code, which imports the **/var/db/scripts/lib/test.xsl** file:

XSLT Syntax

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0">
  <xsl:import href="../lib/test.xsl"/>
  ...
</xsl: stylesheet>
```

SLAX Syntax

```
version 1.0;
import "../lib/test.xsl";
```

Related Documentation

- Storing Scripts in Flash Memory on page 96

CHAPTER 8

Configuring a Remote Source for Scripts

Overview of Updating Scripts from a Remote Source

You can update the scripts on a device running Junos OS by retrieving a copy of them from a remote machine (which can be another device running Junos OS or a regular networked computer). This eases file management, because it enables you to update a script in a single location and have devices update their copies from that location. Each device continues to use its locally stored scripts, only updating a script when you issue the appropriate configuration mode command.

For each script, you can define a remote location that houses the master copy of the script, by specifying its URL with the **source** statement at the hierarchy level where you configured the script. When you then issue the **set refresh** configuration mode command for a script, the device running Junos OS updates its local copy by retrieving the remote master copy from that URL.

You can also store a copy of a particular script at a remote location other than the master source. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems. When you issue the **set refresh-from** command for the script, you specify the URL for the remote script as an option to the command.

You can use the **set refresh** and **set refresh-from** commands to update either an individual commit script or all enabled commit scripts on the device. When you issue the **set refresh** or **set refresh-from** command, the switch, router, or security device immediately attempts to connect to the appropriate remote source for each script. If successful, the device updates the local script with the remote source. If a problem occurs, a set of error messages is returned.

Issuing the **set refresh** or **set refresh-from** command does not add the **refresh** and **refresh-from** statements to the configuration. In other words, the **set** command behaves differently for these statements than for others, It behaves like an operational mode command by executing an operation, instead of adding a statement to the configuration.

If a device has dual Routing Engines and you want the script to be updated on both Routing Engines, you must include the **refresh** or **refresh-from** statements in the configuration of both Routing Engines. The **commit synchronize** command does not cause the **refresh** or **refresh-from** statement to update scripts on both Routing Engines.

The **refresh** and **refresh-from** statements are mutually exclusive.

CAUTION: For commit scripts, we recommend that you do not automate the update function by including the **refresh** statement as a commit script change element. Even though this might seem like a good way to ensure that the most current commit script is always used, we recommend against it for the following reasons:

- Automated update means that the network must be operational for the commit operation to succeed. If the network goes down after you make a configuration error, you cannot recover quickly.
- If multiple commit scripts need to be updated during each commit operation, the network response time can slow down.
- Automated update is always the last action performed during a commit operation. Consequently, the updated commit script executes only during the next commit operation. This is because commit scripts are applied to the candidate configuration before the software copies any persistent changes generated by the scripts to the candidate configuration. In contrast, if you perform the update operation manually, the updated commit script takes effect as expected, that is, immediately after you commit the **refresh** statement in the configuration.
- If you automate the update operation, the **refresh-from** statement has no effect, because the **refresh-from** URL conflicts with and is overridden by the **source** statement URL. For information about the **refresh-from** statement, see "Using an Alternate Source Location for a Script" on page 105.

Related Documentation

- Using a Master Source Location for a Script on page 100
- Using an Alternate Source Location for a Script on page 105

Using a Master Source Location for a Script

Configuring and Refreshing from the Master Source for a Script

You can store a master copy of each script in a central repository. This eases file management because you can make changes to the master script in one place and then

update the copy on each device where the script is currently enabled. This section discusses the following concepts:

- Configuring the Master Source for a Script on page 101
- Updating a Script from the Master Source on page 101

Configuring the Master Source for a Script

To specify the location of the master source for a script, include the **source** statement at the hierarchy level where the script is configured. Including the **source** statement in the configuration does not affect the local copy of the script until you issue the **set refresh** command. At that point, the master copy is retrieved from the specified URL and overwrites the local copy.

The hierarchy location for the **source** statement depends on the script type and filename.

commit script—Include the **source** statement at the **[edit system scripts commit file *filename*]** hierarchy level.

```
[edit system scripts commit file filename]
user@R1# set source url
```

op script—Include the **source** statement at the **[edit system scripts op file *filename*]** hierarchy level.

```
[edit system scripts op file filename]
user@R1# set source url
```

event script—Include the **source** statement at the **[edit event-options event-script file *filename*]** hierarchy level.

```
[edit event-options event-script file filename]
user@R1# set source url
```

Where

- ***filename***—Name of the script.
- ***url***—URL of the script's master source file. Specify the source as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

Updating a Script from the Master Source

If you configure a master source for one or more scripts on a device, you can refresh the scripts on that device by issuing the **set refresh** configuration mode command. You can update a single script or all scripts of a given script type that have a master source location configured.

When you issue the **set refresh** command, the switch, router, or security device immediately attempts to connect to the machine that houses the master source for the script file and retrieve a copy of the file. The master copy overwrites the local script stored in the scripts directory on the device. If a master source is not defined for a script, that script is not updated and a warning is issued. For commit scripts, the updated commit script is executed when you next issue the **commit** command.

The update operation occurs as soon as you issue the **set refresh** command. The **refresh** statement is not added to the configuration. Thus the **set refresh** command behaves like an operational mode command, instead of adding a statement to the configuration.

If the device has dual Routing Engines and you want to update a script on both Routing Engines, you must issue the **set refresh** command on each Routing Engine separately. The **commit synchronize** command does not cause the **refresh** statement to update scripts on both Routing Engines.

To update a single script from its master source, issue the **set refresh** command at the hierarchy level where the script is configured. The **source** statement specifying the master source location must already be configured.

commit script—Issue the **set refresh** command at the **[edit system scripts commit file *filename*]** hierarchy level.

```
[edit system scripts commit file filename]
user@R1# set refresh
```

op script—Issue the **set refresh** command at the **[edit system scripts op file *filename*]** hierarchy level.

```
[edit system scripts op file filename]
user@R1# set refresh
```

event script—Issue the **set refresh** command at the **[edit event-options event-script file *filename*]** hierarchy level.

```
[edit event-options event-script file filename]
user@R1# set refresh
```

Where ***filename*** is the name of the script.

To update all enabled scripts of a given script type from their master source files, issue the **set refresh** command at the hierarchy level for that script type.

commit scripts—Issue the **set refresh** command at the **[edit system scripts commit]** hierarchy level:

```
[edit system scripts commit]
user@R1# set refresh
```

op scripts—Issue the **set refresh** command at the **[edit system scripts op]** hierarchy level:

```
[edit system scripts op]
user@R1# set refresh
```

event scripts—Issue the **set refresh** command at the **[edit event-options event-script]** hierarchy level:

```
[edit event-options event-script]
user@R1# set refresh
```


Example: Configuring and Refreshing from the Master Source for a Script

The following example configures a master source file for an op script on a device running Junos OS. The remote source is defined as an HTTP URL. The example uses the master source to update the local copy of the script on the device.

Requirements

- Routing, switching, or security device running Junos OS.

Overview

You can store a master copy of each script in a central repository. You can make changes to the master script in one place and then update the local copy of the script on devices where the script is enabled.

This example enables the op script **iso.xsl** on a device running Junos OS and then configures a master source location for the script. The remote source for the **iso.xsl** file is the HTTP URL http://my.example.com/pub/scripts/iso.xsl.

Once you configure the master source location, you refresh the local script by issuing the **set refresh** configuration mode command at the hierarchy level where you configured the script. In this example, you would issue the **set refresh** command at the **[edit system scripts op file iso.xsl]** hierarchy level.

Configuration

Step-by-Step Procedure

To download, enable, and configure the master source location for the script:

1. Copy the script to the **/var/db/scripts/op/** directory on the device.
2. In configuration mode, configure the **file** statement to enable the **iso.xsl** script.

   ```
   [edit system scripts op]
   user@R1# set file iso.xsl
   ```

3. To configure the master source for the **iso.xsl** file, include the **source** statement and source location at the **[edit system scripts op file iso.xsl]** hierarchy level.

   ```
   [edit system scripts op file iso.xsl]
   user@R1# set source http://my.example.com/pub/scripts/iso.xsl
   ```

4. Issue the **commit and-quit** command to commit the configuration and exit to operational mode.

   ```
   [edit]
   user@R1# commit and-quit
   ```

Results

```
system {
  scripts {
    op {
      file iso.xsl {
        source http://my.example.com/pub/scripts/iso.xsl;
      }
    }
  }
}
```

Verifying the Script

Purpose Verify that the script is on the device and enabled in the configuration.

Action Issue the **file list** operational mode command to view the files in the specified directory. The **detail** option provides additional information such as permissions, file size, and modified date.

```
user@R1> file list /var/db/scripts/op detail

/var/db/scripts/op:
total 128
-rw-r--r--  1 root  admin  13897 Feb 10  2011 iso.xsl
...
```

Issue the **show configuration system scripts op** operational mode command to list the op scripts currently enabled on the device.

```
user@R1> show configuration system scripts op
file iso.xsl
```

Refreshing the Script from the Master Source

Step-by-Step Procedure To refresh the local copy of the script from the master source file:

1. In configuration mode, issue the **set refresh** command at the **[edit system scripts op file iso.xsl]** hierarchy level.

   ```
   [edit system scripts op file iso.xsl]
   user@R1# set refresh
   ```

Verification

Verifying the Updated Script

Purpose After refreshing the script, verify that the local copy is updated.

Action Issue the **file list** operational mode command with the **detail** option to view the files in the specified directory. Verify that the modified date reflects the refreshed version.

```
user@R1> file list /var/db/scripts/op detail

/var/db/scripts/op:
total 128
-rw-r--r--  1 root  admin  14128 May 26  2011 iso.xsl
...
```

Related Documentation

- Using an Alternate Source Location for a Script on page 105
- refresh (Commit Scripts)
- refresh (Op Scripts)
- refresh (Event Scripts)

Using an Alternate Source Location for a Script

Refreshing a Script from an Alternate Location

In addition to updating a script from the master source defined by the **source** statement at the **[edit event-options event-script file *filename*]** hierarchy level, you also can update a script from an alternate location. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems.

When you issue the **set refresh-from** command, the switch, router, or security device immediately attempts to connect to the machine that houses the alternate source for the script file and retrieve a copy of the file. The copy overwrites the local script stored in the scripts directory on the device. If a copy of the source is not available, that script is not updated and a warning is issued. For commit scripts, the updated commit script is executed when you next issue the **commit** command.

The update operation occurs as soon as you issue the **set refresh-from** command. The **refresh-from** statement is not added to the configuration. Thus the **set refresh-from** command behaves like an operational mode command, instead of adding a statement to the configuration.

If a device has dual Routing Engines and you want the script to be updated on both Routing Engines, you must issue the **set refresh-from** command on each Routing Engine separately. The **commit synchronize** command does not cause the **refresh-from** statement to update scripts on both Routing Engines.

When you issue the **set refresh-from** command, Junos OS creates a folder in the **/var/tmp** directory. This folder is used for file transfer. After the transfer and refresh operations are complete, Junos OS deletes the temporary folder.

To update a single script from the alternate source, issue the **set refresh-from** command at the hierarchy level where the script is configured, and specify the location of the remote file.

commit script—Issue the **set refresh-from** command at the **[edit system scripts commit file *filename*]** hierarchy level.

```
[edit system scripts commit file filename]
user@R1# set refresh-from url
```

op script—Issue the **set refresh-from** command at the **[edit system scripts op file *filename*]** hierarchy level.

```
[edit system scripts op file filename]
user@R1# set refresh-from url
```

event script—Issue the **set refresh-from** command at the **[edit event-options event-script file *filename*]** hierarchy level.

```
[edit event-options event-script file filename]
user@R1# set refresh-from url
```

To update all enabled scripts of a given script type from an alternate source, issue the **set refresh-from** command at the hierarchy level for that script type, and specify the location of the remote directory that houses the scripts.

commit script—Issue the **set refresh-from** command at the **[edit system scripts commit]** hierarchy level.

```
[edit system scripts commit]
user@R1# set refresh-from url
```

op script—Issue the **set refresh-from** command at the **[edit system scripts op]** hierarchy level.

```
[edit system scripts op]
user@R1# set refresh-from url
```

event script—Issue the **set refresh-from** command at the **[edit event-options event-script]** hierarchy level.

```
[edit event-options event-script]
user@R1# set refresh-from url
```

Where

url—URL of the remote script or directory. Specify the source as a Hypertext Transfer Protocol (HTTP) URL, FTP URL, or secure copy (scp)-style remote file specification.

Example: Refreshing a Script from an Alternate Source

The following example uses an alternate source location to update the local copy of the script on a device running Junos OS. The remote source is defined as an HTTP URL.

Requirements

- Routing, switching, or security device running Junos OS.

Overview

You can update a script from a location other than that of the master source. This is convenient when, for example, the master source cannot be accessed due to network issues or other problems. You can refresh a single script or all scripts of a given type from the alternate location.

This example enables the op script **iso.xsl** on a device running Junos OS and then refreshes the script from a location other than the master source location. The remote source for the **iso.xsl** file is the HTTP URL http://my.example.com/pub/scripts2/iso.xsl.

You refresh the local script by issuing the **set refresh-from** configuration mode command at the hierarchy level where you configured the script. In this example, you would issue the **set refresh-from** command at the **[edit system scripts op file iso.xsl]** hierarchy level.

Configuration

Step-by-Step Procedure

To download and enable the script:

1. Copy the script to the **/var/db/scripts/op/** directory on the device.
2. In configuration mode, configure the **file** statement to enable the **iso.xsl** script.

   ```
   [edit system scripts op]
   user@R1# set file iso.xsl
   ```

3. Issue the **commit and-quit** command to commit the configuration and exit to operational mode.

   ```
   [edit]
   user@R1# commit and-quit
   ```

Results

```
system {
  scripts {
    op {
      file iso.xsl;
    }
  }
}
```

Verifying the Script

Purpose Verify that the script is on the device and enabled in the configuration.

Action Issue the **file list** operational mode command to view the files in the specified directory. The **detail** option provides additional information such as permissions, file size, and modified date.

```
user@R1> file list /var/db/scripts/op detail
```

```
/var/db/scripts/op:
total 128
-rw-r--r--  1 root  admin  13897 Feb 10  2011 iso.xsl
...
```

Issue the **show configuration system scripts op** operational mode command to list the op scripts currently enabled on the device.

```
user@R1> show configuration system scripts op
file iso.xsl
```

Refreshing the Script from the Alternate Location

Step-by-Step Procedure

To refresh the local copy of the script from the alternate location:

1. In configuration mode, issue the **set refresh-from** command at the **[edit system scripts op file iso.xsl]** hierarchy level.

```
[edit system scripts op file iso.xsl]
user@R1# set refresh-from http://my.example.com/pub/scripts2/iso.xsl
```

Verification

Verifying the Updated Script

Purpose

After refreshing the script, verify that the local copy is updated.

Action

Issue the **file list** operational mode command with the **detail** option to view the files in the specified directory. Verify that the modified date reflects the refreshed version.

```
user@R1> file list /var/db/scripts/op detail
```

```
/var/db/scripts/op:
total 128
-rw-r--r--  1 root  admin  14128 May 26  2011 iso.xsl
...
```

Related Documentation

- Using a Master Source Location for a Script on page 100
- refresh-from (Commit Scripts)
- refresh-from (Op Scripts)
- refresh-from (Event Scripts)

CHAPTER 9

Configuring the Session Protocol for Scripts

Specifying the Session Protocol for Connections Using Junos Automation Scripts

Session Protocol in Junos Automation Scripts Overview

The Junos XML management protocol is a Juniper Networks proprietary protocol used to request information from and configure devices running Junos OS. The NETCONF XML management protocol is a standard used to request and change configuration information on a routing, switching, or security device. The NETCONF protocol is defined in RFC 4741, *NETCONF Configuration Protocol*, which is available at http://www.ietf.org/rfc/rfc4741.txt .

Starting with Junos OS Release 11.4, the **jcs:open( )** function includes the option to create a session either with the Junos XML protocol server on devices running Junos OS or with the NETCONF server on devices where NETCONF service over SSH is enabled. Previously, the function supported only sessions with the Junos XML protocol server on devices running Junos OS. The additional support for NETCONF sessions permits automation scripts to configure and manage devices in a multi-vendor environment.

The **jcs:open( )** function supports the following session protocol types:

- **junoscript**—Session with the Junos XML protocol server on a routing, switching, or security device running Junos OS. This session type supports the operations defined in the Junos XML protocol and the Junos XML API, which are used to configure devices running Junos OS or to request information about the device configuration or operation. This is the default session type.
- **netconf**—Session with the NETCONF XML protocol server on a routing, switching, or security device over an SSHv2 connection. The device to which the connection is made must be enabled for NETCONF service over SSH. NETCONF over SSH is described in

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

- **junos-netconf**—Proprietary session with the NETCONF XML protocol server over an SSHv1 connection on a routing, switching, or security device running Junos OS.

The NETCONF server on a device running Junos OS has the additional capabilities defined in **http://xml.juniper.net/netconf/junos/1.0** . The NETCONF server on these devices supports NETCONF XML protocol operations, most Junos XML protocol operations, and the tag elements defined in the Junos XML API. For **netconf** and **junos-netconf** sessions with devices running Junos OS, you should use only native NETCONF XML protocol operations and the extensions available in the Junos XML protocol for configuration functions as documented in the *NETCONF XML Management Protocol Guide*.

The syntax for the **jcs:open()** function when specifying a session protocol is:

SLAX Syntax

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

XSLT Syntax

```
<xsl:variable name="connection" select="jcs:open(remote-hostname,session-options)"/>
```

The *session-options* parameter is an XML node-set that specifies the session type and connection parameters. The session type is one of three values: **junoscript**, **netconf**, or **junos-netconf**. If you do not specify a session type, the default is **junoscript**, which opens a session with the Junos XML protocol server only on devices running Junos OS. The format of the node-set is:

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

If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script. The **<passphrase>** and **<password>** elements serve the same purpose. If you do not specify a passphrase or password element and it is required for authentication, you should be prompted for one during script execution by the device to which you are connecting.

Optionally, you can specify the server port number for **netconf** and **junos-netconf** sessions. The default NETCONF server port number is 830. If you do not specify a port number for a **netconf** or **junos-netconf** session, **jcs:open()** connects to the NETCONF server using port 830. However, if you specify a port number, **jcs:open()** connects to the given port instead. Specifying a port number has no impact on **junoscript** sessions, which are always established over SSH port 22.

To verify the protocol for a specific connection, call the **jcs:get-protocol(*connection*)** extension function and pass the connection handle as the argument. The function returns “junoscript”, “netconf”, or “junos-netconf”, depending on the session type.

During session establishment with a NETCONF server, the client application and NETCONF server each emit a **<hello>** tag element to specify which operations, or *capabilities,* they

support from among those defined in the NETCONF specification or published as proprietary extensions. In **netconf** and **junos-netconf** sessions, you can retrieve the session capabilities of the NETCONF server by calling the **jcs:get-hello(*connection*)** extension function.

For example, the NETCONF server on a typical device running Junos OS might return the following capabilities:

```
<hello>
  <capabilities>
     <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:validate:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
     </capability>
     <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
     <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>20826</session-id>
</hello>
```

Example: Specifying the Session Protocol for a Connection Using an Automation Script

The following example demonstrates how to specify the session protocol within a Junos automation script when creating a connection with a remote device. Specifically, the example op script establishes a NETCONF session with a remote device running Junos OS, retrieves and prints the NETCONF server capabilities, and then updates and commits the configuration on that device.

Requirements

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

Overview and Script

Starting with Junos OS Release 11.4, the **jcs:open()** function includes the option to create a session either with the Junos XML protocol server on devices running Junos OS or with

the NETCONF server on devices where NETCONF service over SSH is enabled. In the following example, the script creates a connection and establishes a NETCONF session with a remote device running Junos OS. If the connection and session are successfully established, the script updates the configuration on the remote device to add the **ftp** statement to the **[edit system services]** hierarchy level. The script also retrieves and prints the session protocol and the capabilities of the NETCONF server.

The script takes one argument, **remote-host**, which is the IP address or hostname of the remote device. The **arguments** variable is declared at the global level of the script so that the argument name and description are visible in the command-line interface (CLI) when a user requires context-sensitive help.

The variable **netconf** is a node-set that specifies the session protocol and the connection parameters for the remote device. The value of the **<method>** element is set to "netconf" to establish a session with the NETCONF server over an SSHv2 connection. The **<username>** element specifies the username for the connection. If you do not specify a username and it is required for the connection, the script uses the local name of the user executing the script. In this example, the passphrase and port are not specified. If a passphrase is required for authentication, the remote device should prompt for one during script execution. The script establishes the session using the default NETCONF port 830.

If the connection and establishment of the NETCONF session are successful, the script executes remote procedure calls (RPCs). The RPCs contain the tag elements **<lock>**, **<edit-config>**, **<commit>**, and **<unlock>**, which are NETCONF operations to lock, edit, commit, and unlock the candidate configuration. The script stores the RPC for each task in a separate variable. The results for each RPC are also stored separately and parsed for errors. The script only executes each subsequent step if the previous step is successful. For example, if the script cannot lock the configuration, it does not execute the RPCs to edit, commit, or unlock the configuration.

The variable **rpc-edit-config** contains the tag element **<edit-config>**, which is a NETCONF operation to modify a configuration. The child element, **<config>**, includes the modified portion of the configuration that is merged with the candidate configuration on the device. If errors are encountered, the script calls the **copy-of** statement to copy the result tree fragment variable to the results tree so that the error message prints to the CLI during script execution.

SLAX Syntax

```
version 1.0;

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

var $arguments = {
   <argument> {
      <name> "remote-host";
      <description> "device hostname or IP address to which to connect";
   }
}
param $remote-host;
```

```
match / {

  <op-script-results> {

    var $netconf := {
      <method> "netconf";
      <username> "bsmith";
    }

    var $rpc-lock-config = {
      <lock> {
        <target> {
          <candidate>;
        }
      }
    }

    var $rpc-unlock-config = {
      <unlock> {
        <target> {
          <candidate>;
        }
      }
    }

    var $rpc-commit = {
      <commit>;
    }

    var $rpc-edit-config = {
      <edit-config> {
        <target> {
          <candidate>;
        }
        <default-operation> "merge";
        <config> {
          <configuration> {
            <system> {
              <services> {
                <ftp>;
              }
            }
          }
        }
      }
    }

    if ($remote-host = '') {
      <xnm:error> {
        <message> "missing mandatory argument 'remote-host'";
      }
    }
    else {

      var $connection = jcs:open($remote-host, $netconf);
```

```
    if ($connection) {

      /* request protocol and capabilities */
      var $protocol = jcs:get-protocol($connection);
      var $capabilities = jcs:get-hello($connection);

      <output> "\nSession protocol: " _ $protocol _ "\n";
      copy-of $capabilities;

      /* execute rpcs to lock, edit, commit, and unlock config */
      var $lock-reply = jcs:execute($connection, $rpc-lock-config);
      if ($lock-reply/..//rpc-error) {
        copy-of $lock-reply;
      }
      else {
        var $edit-config-reply = jcs:execute($connection, $rpc-edit-config);
        if ($edit-config-reply/..//rpc-error) {
          <output>"Configuration error: "_ $edit-config-reply/..//error-message/.
                _ "\nConfiguration not committed.\n" ;
          copy-of $edit-config-reply;
        }
        else {
          var $commit-reply = jcs:execute($connection, $rpc-commit);
          if ($commit-reply/..//rpc-error) {
            <output>"Commit error or warning: " _ $commit-reply/..//error-message/.;
            copy-of $commit-reply;
          }
        }
        var $unlock-reply = jcs:execute($connection, $rpc-unlock-config);
      }

      expr jcs:close($connection);
    }
    else {
      <output>"\nNo connection - exiting script";
    }

  }
 }
}
```

XSLT Syntax

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

 <xsl:variable name="arguments">
  <argument>
   <name>remote-host</name>
   <description>device hostname or IP address to which to connect</description>
  </argument>
 </xsl:variable>

 <xsl:param name="remote-host"/>
```

```
<xsl:template match="/">
 <op-script-results>
  <xsl:variable name="netconf-temp-1">
   <method>netconf</method>
   <username>bsmith</username>
  </xsl:variable>
  <xsl:variable xmlns:ext="http://xmlsoft.org/XSLT/namespace"
   name="netconf" select="ext:node-set($netconf-temp-1)"/>

  <xsl:variable name="rpc-lock-config">
   <lock>
    <target>
     <candidate/>
    </target>
   </lock>
  </xsl:variable>

  <xsl:variable name="rpc-unlock-config">
   <unlock>
    <target>
     <candidate/>
    </target>
   </unlock>
  </xsl:variable>

  <xsl:variable name="rpc-commit">
   <commit/>
  </xsl:variable>

  <xsl:variable name="rpc-edit-config">
   <edit-config>
    <target>
     <candidate/>
    </target>
    <default-operation>merge</default-operation>
    <config>
     <configuration>
      <system>
       <services>
        <ftp/>
       </services>
      </system>
     </configuration>
    </config>
   </edit-config>
  </xsl:variable>

  <xsl:choose>
   <xsl:when test="$remote-host = ''">
    <xnm:error>
     <message>missing mandatory argument 'remote-host'</message>
    </xnm:error>
   </xsl:when>
   <xsl:otherwise>
    <xsl:variable name="connection" select="jcs:open($remote-host, $netconf)"/>
```

```
<xsl:choose>
 <xsl:when test="$connection">

  <!-- request protocol and capabilities -->
  <xsl:variable name="protocol" select="jcs:get-protocol($connection)"/>
  <xsl:variable name="capabilities" select="jcs:get-hello($connection)"/>
  <output>
   <xsl:value-of select="concat('&#10;Session protocol: ', $protocol, '&#10;')"/>
  </output>
  <xsl:copy-of select="$capabilities"/>

  <!-- execute rpcs -->
  <xsl:variable name="lock-reply"
   select="jcs:execute($connection, $rpc-lock-config)"/>
  <xsl:choose>
   <xsl:when test="$lock-reply/..//rpc-error">
    <xsl:copy-of select="$lock-reply"/>
   </xsl:when>
   <xsl:otherwise>
    <xsl:variable name="edit-config-reply"
     select="jcs:execute($connection, $rpc-edit-config)"/>
    <xsl:choose>
     <xsl:when test="$edit-config-reply/..//rpc-error">
      <output>
       <xsl:value-of select="concat('Configuration error: ',
        $edit-config-reply/..//error-message/.,
        '&#10;Configuration not committed.&#10;')"/>
      </output>
      <xsl:copy-of select="$edit-config-reply"/>
     </xsl:when>
     <xsl:otherwise>
      <xsl:variable name="commit-reply"
       select="jcs:execute($connection, $rpc-commit)"/>
      <xsl:if test="$commit-reply/..//rpc-error">
       <output>
        <xsl:value-of select="concat('Commit error or warning: ',
         $commit-reply/..//error-message/.)"/>
       </output>
       <xsl:copy-of select="$commit-reply"/>
      </xsl:if>
     </xsl:otherwise>
    </xsl:choose>
    <xsl:variable name="unlock-reply" select="jcs:execute($connection,
     $rpc-unlock-config)"/>
   </xsl:otherwise>
  </xsl:choose>

  <xsl:value-of select="jcs:close($connection)"/>
 </xsl:when>
 <xsl:otherwise>
  <output>No connection - exiting script</output>
 </xsl:otherwise>
</xsl:choose>
</xsl:otherwise>
</xsl:choose>
```

```
    </op-script-results>
  </xsl:template>
</xsl:stylesheet>
```

Configuration

Step-by-Step Procedure

To download, enable, and test the script:

1. Copy the XSLT or SLAX script into a text file, name the file **netconf-session.xsl** or **netconf-session.slax** as appropriate, and copy it to the **/var/db/scripts/op/** directory on the device.
2. In configuration mode, include the **file** statement at the **[edit system scripts op]** hierarchy level and **netconf-session.xsl** or **netconf-session.slax** as appropriate.

   ```
   [edit system scripts op]
   bsmith@local-host# set file netconf-session.(slax | xsl)
   ```

3. Issue the **commit and-quit** command.

   ```
   [edit]
   bsmith@local-host# commit and-quit
   ```

4. Execute the op script on the local device by issuing the **op netconf-session** operational mode command and include any necessary arguments.

   In this example, the user, bsmith, is connecting to the remote device, fivestar. The remote device has dual routing engines, so the **commit** operation returns a warning that the **commit synchronize** command should be used to commit the new candidate configuration to both routing engines.

   ```
   bsmith@local-host> op netconf-session remote-host fivestar

   bsmith@fivestar's password:
   Session protocol: netconf
   Commit error or warning:
   graceful-switchover is enabled, commit synchronize should be used
   ```

Verification

Confirm that the device is working properly.

- Verifying Op Script Execution on page 117
- Verifying the Configuration Changes on page 119

Verifying Op Script Execution

Purpose Verify that the script behaves as expected.

Action Review the script output in the CLI and in the op script log file. Take particular note of any errors that occurred during execution. The default op script log file is **/var/log/op-script.log**. If the log file is significantly lengthy, limit the display by appending the **| last** ***number-of-lines*** option to the **file show** command and specify the number of lines to print to the CLI. The output within the **<op-script-results>** element is relevant to the script execution.

```
bsmith@local-host> file show /var/log/op-script.log | last 100
...output omitted for brevity...
<op-script-results xmlns:junos="http://xml.juniper.net/junos/*/junos"
xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm" xml
ns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0"
xmlns:ext="http://xmlsoft.org/XSLT/namespace">
<output>
Session protocol: netconf
</output>
<hello>
  <capabilities>
   <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
   <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
   </capability>
   <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
   </capability>
  <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>

   <capability>
      urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
   </capability>
   <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
   <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>29087</session-id>
</hello>
  <output>Commit error or warning:
graceful-switchover is enabled, commit synchronize should be used
</output>
  <rpc-error>
<error-severity>warning</error-severity>
<error-message>
graceful-switchover is enabled, commit synchronize should be used
</error-message>
</rpc-error>
  <ok/>
</op-script-results>
```

You can also obtain more descriptive script output on a device running Junos OS by including the **| display xml** option when you execute an op script.

```
bsmith@local-host> op netconf-session remote-host fivestar | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/11.4D0/junos">
  <output>
      Session protocol: netconf
  </output>
  <hello>
   <capabilities>
     <capability>
       urn:ietf:params:xml:ns:netconf:base:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:candidate:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0
     </capability>
     <capability>
       urn:ietf:params:xml:ns:netconf:capability:validate:1.0
     </capability>
     <capability>
```

```
        urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
      </capability>
      <capability>
        http://xml.juniper.net/netconf/junos/1.0
      </capability>
      <capability>
        http://xml.juniper.net/dmi/system/1.0
      </capability>
    </capabilities>
    <session-id>
        29087
    </session-id>
  </hello>
  <output>
      Commit error or warning:
      graceful-switchover is enabled, commit synchronize should be used
  </output>
  <rpc-error>
      <error-severity>
          warning
      </error-severity>
      <error-message>
          graceful-switchover is enabled, commit synchronize should be used
      </error-message>
  </rpc-error>
  <ok/>
</op-script-results>
<cli>
  <banner></banner>
</cli>
</rpc-reply>
```

Meaning This example creates a NETCONF session on a remote device running Junos OS. The capabilities of the NETCONF server include both standard NETCONF operations and Juniper Networks proprietary extensions, which are defined in **http://xml.juniper.net/netconf/junos/1.0** and **http://xml.juniper.net/dmi/system/1.0** . The RPC results for the **commit** operation include one warning, but the commit operation is still successful.

Verifying the Configuration Changes

Purpose Verify that the commit was successful by viewing the configuration change and the commit log on the remote device.

Action On the remote device, execute the **show configuration system services** operational mode command to view the **[edit system services]** hierarchy level of the configuration. If the script is successful, the configuration includes the **ftp** statement.

```
bsmith@fivestar> show configuration system services
ftp;
netconf {
    ssh;
}
```

Additionally, you can review the commit log. On the remote device, execute the **show system commit** operational mode command to view the commit log. In this example, the

log confirms that bsmith committed the candidate configuration in a NETCONF session at the given date and time.

```
bsmith@fivestar> show system commit
0   2011-07-11 12:04:01 PDT by bsmith via netconf
1   2011-07-08 15:16:33 PDT by root via cli
```

Troubleshooting

Troubleshooting Connection Errors

Problem The script generates the following error message:

```
hello packet:1:(0) Document is empty
hello packet:1:(0) Start tag expected, '<' not found
error: netconf: could not read hello
error: did not receive hello packet from server
error: Error in creating the session with "fivestar" server
No connection - exiting script
```

Potential causes for the connection error include:

- The device or interface to which you are connecting is down or unavailable.
- The script argument for the IP address or DNS name of the remote device is incorrect.
- The connection timeout value was exceeded before establishing the connection.
- The user authentication for the remote device is not valid or is entered incorrectly.
- You are trying to establish a NETCONF session, and NETCONF over SSH is not enabled on the device where the NETCONF server resides, or it is enabled on a different port.

Solution Ensure that the remote device is up and running and that the user has access to the device. Also verify that you supplied the correct argument for the IP address or DNS name of the remote device when executing the script.

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

```
bsmith@fivestar> show configuration system services

netconf {
    ssh;
}
```

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

```
[edit]
bsmith@fivestar# set system services netconf ssh
bsmith@fivestar# commit
```


If the **netconf** configuration hierarchy specifies a port other than the default port, include the port number in the XML node-set that you pass to the **jcs:open( )** function. For example, the following device is configured for NETCONF over SSH on port 12345:

```
bsmith@fivestar> show configuration system services
netconf {
    ssh {
       port 12345;
    }
}
```

To create a NETCONF session on the alternate port, include the new port number in the XML node-set.

```
var $netconf := {
  <method> "netconf";
  <username> "bsmith";
  <port> "12345";
}
var $connection = jcs:open($remote-host, $netconf);
...
```

Troubleshooting Configuration Lock Errors

Problem The script generates one of the following error messages:

```
configuration database locked by:
  root terminal p0 (pid 24113) on since 2011-07-11 11:48:06 PDT, idle
00:07:59

Users currently editing the configuration:
  root terminal p1 (pid 24279) on since 2011-07-11 12:28:30 PDT
      {master}[edit]

configuration database modified
```

Solution Another user currently has a lock on the candidate configuration or has modified the candidate configuration but has not yet committed the configuration. Wait until the lock is released, and then execute the program.

Troubleshooting Configuration Syntax Errors

Problem The following error message prints to the CLI:

```
Configuration error: syntax error
Configuration not committed.
```

Examine the result tree for additional information. In this case, the result tree shows the following error message:

```
<rpc-error>
   <error-severity>
      error
   </error-severity>
   <error-info>
      <bad-element>
```

```
            ftp2
         </bad-element>
      </error-info>
      <error-message>
         syntax error
      </error-message>
   </rpc-error>
```

Solution The **<bad-element>** tag element indicates that the configuration statement is not valid. Correct the configuration hierarchy and run the script. In this example error, the user entered the tag **<ftp2>** instead of **<ftp>**. Since that is not an acceptable element in the configuration, the NETCONF server returns an error.

Related Documentation

- Session Protocol in Junos Automation Scripts Overview on page 109
- *Junos XML Management Protocol Guide*
- *NETCONF XML Management Protocol Guide*
- jcs:get-hello( ) Function on page 19
- jcs:get-protocol( ) Function on page 20
- jcs:open( ) Function on page 22

CHAPTER 10

SLAX Statements

apply-templates

Syntax apply-templates *expression*;

Description Apply one or more templates, according to the value of the node-set expression. If a node-set expression is not specified, the script recursively processes all child nodes of the current node. If a node-set expression is specified, the processor only applies templates to the child elements that match the node-set expression. The **template** statement dictates which elements are transformed according to which template. The templates that are applied are passed the parameters specified by the **with** statement within the **apply-templates** statement block.

Attributes *expression*—(Optional) Selects the nodes to which the processor applies templates. By default, the processor applies templates to the child nodes of the current node.

SLAX Example

```
match configuration {
  apply-templates system/host-name;
}
```

XSLT Equivalent

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

Usage Examples See Example: Adding a Final then accept Term to a Firewall and Example: Preventing Import of the Full Routing Table.

Related Documentation

- SLAX Templates Overview on page 77
- call on page 125
- match on page 130
- mode on page 131
- priority on page 133
- template on page 134
- with on page 137

call

Syntax

```
call template-name (parameter-name = value) {
  /* code */
}
```

Description

Call a named template. You can pass parameters into the template by including a comma-separated list of parameters, with the parameter name and an optional equal sign (=) and value expression. If a value is not specified, the current value of the parameter is passed to the template.

You can declare additional parameters inside the code block using the **with** statement.

Attributes

template-name—Specifies the name of the template to call.

SLAX Example

```
match configuration {
  var $name-servers = name-servers/name;
  call temp();
  call temp($name-servers, $size = count($name-servers));
  call temp() {
    with $name-servers;
    with $size = count($name-servers);
  }

  template temp($name-servers, $size = 0) {
    <output> "template called with size " _ $size;
  }
}
```

XSLT Equivalent

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

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

Usage Examples

See Example: Requiring and Restricting Configuration Statements, Example: Imposing a Minimum MTU Setting, and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- SLAX Templates Overview on page 77
- apply-templates on page 124
- match on page 130
- mode on page 131
- priority on page 133
- template on page 134
- with on page 137

else

Syntax

```
if (expression) {
    /* code */
}
else {
  /* code */
}
```

Description

Include a default set of instructions that are processed if the preceding **if** and **else if** statements evaluate to FALSE.

SLAX Example

```
if (starts-with(name, "fe-")) {
  if (mtu < 1500) {
    /* Select the Fast Ethernet interfaces with low MTUs */
  }
}
else {
  if (mtu > 8096) {
    /* Select the non-Fast Ethernet interfaces with high MTUs */
  }
}
```

XSLT Equivalent

```
<xsl:choose>
  <xsl:when select="starts-with(name, 'fe-')">
    <xsl:if test="mtu &lt; 1500">
      <!-- Select with Fast Ethernet interfaces with low MTUs -->
    </xsl:if>
  </xsl:when>
  <xsl:otherwise>
    <xsl:if test="mtu &gt; 8096">
      <!-- Select the non-Fast Ethernet interfaces with high MTUs -->
    </xsl:if>
  </xsl:otherwise>
</xsl:choose>
```

Usage Examples

See Example: Configuring Dual Routing Engines and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- SLAX Statements Overview on page 85
- else if on page 127
- for-each on page 128
- if on page 129

else if

Syntax

```
if (expression) {
    /* code */
}
else if (expression) {
    /* code */
}
```

Description

Include instructions that are processed if the expression defined in the preceding **if** statement evaluates to FALSE and the expression defined in the **else if** statement evaluates to TRUE. Multiple **else if** statements can be included, but the processor only executes the instructions contained in the first **else if** statement whose expression evaluates to TRUE. All subsequent **else if** statements are ignored.

SLAX Example

```
var $description2 = {
  if (description) {
    expr description;
  }
  else if (../description) {
    expr ../description;
  }
  else {
    expr "no description found";
  }
}
```

XSLT Equivalent

```
<xsl:variable name="description2">
  <xsl:choose>
    <xsl:when test="description">
      <xsl:value-of select="description"/>
    </xsl:when>
    <xsl:when test="../description">
      <xsl:value-of select="../description"/>
    </xsl:when>
    <xsl:otherwise>unknown</xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

Usage Examples

See Example: Configuring Dual Routing Engines and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- SLAX Statements Overview on page 85
- else on page 126

- for-each on page 128
- if on page 129

for-each

Syntax

```
for-each (expression) {
  /* code */
}
```

Description

Include a looping mechanism that repeats script processing for each XML element in the specified node-set. The element nodes are selected by the value of the XPath expression. Each of the nodes is then processed according to the instructions contained in the **for-each** code block.

The statement consists of the **for-each** keyword, the parentheses-delimited XPath expression, and a curly braces-delimited block. The SLAX **for-each** statement functions like the **<xsl:for-each>** element.

Attributes

expression—Specifies an XPath expression that selects the nodes to be processed.

SLAX Example

```
for-each ($inventory/chassis/chassis-module/
      chassis-sub-module[part-number == '750-000610']) {
  <message> "Down rev PIC in " _ ../name _ ", " _ name _ ": " _ description;
}
```

XSLT Equivalent

```
<xsl:for-each select="$inventory/chassis/chassis-module/
      chassis-sub-module[part-number == '750-000610']">
  <message>
    <xsl:value-of select="concat('Down rev PIC in ', ../name, ', ', name, ': ',
        description)"/>
  </message>
</xsl:for-each>
```

Usage Examples

See Example: Requiring and Restricting Configuration Statements, Example: Imposing a Minimum MTU Setting, Example: Limiting the Number of E1 Interfaces, Example: Adding T1 Interfaces to a RIP Group, Example: Configuring Administrative Groups for LSPs, and Example: Configuring Dual Routing Engines.

Related Documentation

- SLAX Statements Overview on page 85
- XPath Overview on page 55
- else on page 126
- else if on page 127
- if on page 129
- xsl:for-each on page 149

if

Syntax

```
if (expression) {
  /* code */
}
else if (expression) {
    /* code */
}
else {
  /* code */
}
```

Description Include a conditional construct that causes instructions to be processed if the Boolean expression evaluates to TRUE.

Optionally, you can include multiple **else if** statements following an **if** statement to perform additional conditional tests if the expression in the **if** statement evaluates to FALSE. Multiple **else if** statements can be included, but the processor only executes the instructions contained in the first **else if** statement whose expression evaluates to TRUE; all subsequent **else if** statements are ignored. The optional **else** statement includes a default set of instructions that are processed if the expressions defined in all associated **if** and **else if** statements evaluate to FALSE.

Attributes *expression*—Specifies the expression to evaluate.

SLAX Example

```
var $description2 = {
  if (description) {
    expr description;
  }
  else if (../description) {
    expr ../description;
  }
  else {
    expr "no description found";
  }
}
```

XSLT Equivalent

```
<xsl:variable name="description2">
  <xsl:choose>
    <xsl:when test="description">
       <xsl:value-of select="description"/>
    </xsl:when>
    <xsl:when test="../description">
      <xsl:value-of select="../description"/>
    </xsl:when>
    <xsl:otherwise>unknown</xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

Usage Examples See Example: Configuring Dual Routing Engines, Example: Preventing Import of the Full Routing Table, and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- SLAX Statements Overview on page 85
- else on page 126
- else if on page 127
- for-each on page 128

match

Syntax

```
match expression {
statements;
}
```

Description Declare a template that contains rules to apply when a specified node is matched. The **match** statement associates the template with an XML element. The **match** statement can also be used to define a template for a whole branch of the XML document For example, **match /** matches the root element of the document.

Attributes *expression*—XPath expression specifying the nodes to which to apply the template.

SLAX Example

```
match host-name {
  <hello> .;
}
```

XSLT Equivalent

```
<xsl:template match="host-name">
  <hello>
    <xsl:value-of select="."/>
  </hello>
</xsl:template>
```

Usage Examples Example: Adding a Final then accept Term to a Firewall, Example: Configuring Dual Routing Engines, and Example: Preventing Import of the Full Routing Table.

Related Documentation

- apply-templates on page 124
- call on page 125
- mode on page 131
- priority on page 133
- template on page 134
- with on page 137

mode

Syntax mode *qualified-name*;

Description Indicate the mode in which a template needs to be applied for the template to be used. If templates are applied in the specified mode, the **match** statement is used to determine whether the template can be used with the particular node. If more than one template matches a node in the specified mode, the priority statement determines which template is used. The highest priority wins. If no priority is specified explicitly, the priority of a template is determined by the **match** statement.

This statement is comparable to the **mode** attribute of the **<xsl:template>** element. You can include this statement inside a SLAX **match** or **apply-templates** statement.

SLAX Example

```
match * {
  mode "one";
  <one> .;
}

match * {
  mode "two";
  <two> string-length(.);
}

match / {
  apply-templates version {
    mode "one";
  }
  apply-templates version {
    mode "two";
  }
}
```

XSLT Equivalent

```
<xsl:template match="*" mode="one">
  <one>
    <xsl:value-of select="."/>
  </one>
</xsl:template>

<xsl:template match="*" mode="two">
  <two>
    <xsl:value-of select="string-length(.)"/>
  </two>
</xsl:template>

<xsl:template match="/">
  <xsl:apply-templates select="version" mode="one"/>
  <xsl:apply-templates select="version" mode="two"/>
</xsl:template>
```

Usage Examples See Example: Adding a Final then accept Term to a Firewall.

Related Documentation

- apply-templates on page 124
- call on page 125
- match on page 130
- priority on page 133
- template on page 134
- with on page 137
- xsl:template on page 154

param

Syntax

```
param $name=value;
```

Description

Declare a parameter for a template or for the style sheet as a whole. Template parameters declared with the **param** statement must be placed inside the template code block. A global parameter, the scope of which is the entire style sheet, must be declared at the top level of the style sheet. You can include an initial value by following the parameter name with an equal sign (**=**) and a value expression. A parameter whose value is set by Junos OS at script initialization must be defined as a global parameter.

In SLAX, parameter and variable names are declared and accessed using the dollar sign (**$**). This is unlike the **name** attribute of **<xsl:variable>** and **<xsl:parameter>** elements, which do not include the dollar sign in the declaration.

Attributes

$*name*—Defines the name of the parameter.

value—Defines the default value for the parameter, which is used if the person or client application that executes the script does not explicitly provide a value.

SLAX Example

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

XSLT Equivalent

```
<xsl:param name="vrf"/>
<xsl:param name="dot" select="."/>
```

Usage Examples

See Example: Requiring and Restricting Configuration Statements, Example: Imposing a Minimum MTU Setting, Example: Limiting the Number of E1 Interfaces, Example: Limiting the Number of ATM Virtual Circuits, and Example: Preventing Import of the Full Routing Table.

Related Documentation

- SLAX Parameters Overview on page 81
- SLAX Templates Overview on page 77
- template on page 134
- var on page 135
- with on page 137

priority

Syntax priority *number*;

Description If more than one template matches a node in the specified mode, this statement determines which template is used. The highest priority wins. If no priority is specified explicitly, the priority of a template is determined by the **match** statement.

This statement is comparable to the **priority** attribute of the **<xsl:template>** element. You can include this statement inside a SLAX **match** statement.

SLAX Example

```
match * {
  priority 10;
  <output> .;
}
```

XSLT Equivalent

```
<xsl:template match="*" priority="10">
  <output>
    <xsl:value-of select="."/>
  </output>
</xsl:template>
```

Usage Examples None of the examples in this manual use this statement.

Related Documentation

- apply-templates on page 124
- call on page 125
- match on page 130
- mode on page 131
- template on page 134
- with on page 137
- xsl:template on page 154

template

Syntax

```
template qualified-name (parameter-name = value) {
  /* code */
}
```

Description

Declare a named template. You can include a comma-separated list of parameter declarations, with the parameter name and an optional equal sign (**=**) and value expression. You can declare additional parameters inside the code block using the **param** statement. You can invoke the template using the **call** statement.

SLAX Example

```
match configuration {
  var $name-servers = name-servers/name;
  call temp();
  call temp($name-servers, $size = count($name-servers));
  call temp() {
    with $name-servers;
    with $size = count($name-servers);
  }

  template temp($name-servers, $size = 0) {
    <output> "template called with size " _ $size;
  }
}
```

XSLT Equivalent

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

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

Usage Examples

See Example: Adding a Final then accept Term to a Firewall and Example: Adding T1 Interfaces to a RIP Group.

Related Documentation

- SLAX Parameters Overview on page 81
- SLAX Templates Overview on page 77

- apply-templates on page 124
- call on page 125
- match on page 130
- mode on page 131
- priority on page 133
- with on page 137

var

Syntax

var $*name*=*value*;

Description

Declare a local or global variable. A variable is global if it is defined outside of any template. Otherwise, it is local. The value of a global variable is accessible anywhere in the style sheet. The scope of a local variable is limited to the template or code block in which it is defined. You initialize a variable by following the variable name with an equal sign (=) and an expression.

Attributes

$*name*—Specifies the name of the variable. After declaration, the variable can be referred to within expressions using this name, including the $ character.

value—Defines the default value for the variable, which is used if the person or client application that executes the script does not explicitly provide a value.

SLAX Example

```
var $vrf;
var $location = $dot/@location;
var $message = "We are in "_ $location _" now.";
```

XSLT Equivalent

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

Usage Examples

See Example: Limiting the Number of E1 Interfaces, Example: Limiting the Number of ATM Virtual Circuits, Example: Configuring Administrative Groups for LSPs, and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- SLAX Variables Overview on page 84
- param on page 132

version

Syntax version 1.0;

Description Specify the version of SLAX that is being used. All SLAX style sheets must begin with a **version** statement.

Version 1.0 uses XML version 1.0 and XSLT version 1.1.

In addition, the **xsl** namespace is implicitly defined as follows:

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

Attributes ***version-number***—Specifies the version of SLAX. Junos OS supports SLAX version 1.0.

SLAX Example

```
version 1.0;
```

XSLT Equivalent

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

Usage Examples Example: Adding a Final then accept Term to a Firewall, Example: Assigning a Classifier, Example: Imposing a Minimum MTU Setting, Example: Changing the Configuration Using an Op Script, and Example: Restarting an FPC Using an Op Script.

Related Documentation

- Required Boilerplate for Commit Scripts
- Required Boilerplate for Event Scripts
- Required Boilerplate for Op Scripts
- SLAX Syntax Rules Overview on page 73

with

Syntax with $*name* = *value*;

Description Specify a parameter to pass into a template. You can use this statement when you apply templates with the **apply-templates** statement or invoke templates with the **call** statement.

Optionally, you can specify a value for the parameter by including an equal sign (=) and a value expression. If no value is specified, the current value of the parameter is passed to the template.

Attributes $*name*—Name of the variable or parameter for which the value is being passed.

value—Value of the parameter being passed to the template.

SLAX Example

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

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

XSLT Equivalent

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

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

Usage Examples See Example: Configuring Dual Routing Engines, Example: Preventing Import of the Full Routing Table, and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- SLAX Parameters Overview on page 81
- SLAX Templates Overview on page 77

CHAPTER 11

Standard XPath and XSLT Functions Used in Automation Scripts

concat( )

Syntax *string* concat(*string*, *string*+)

Description Return the concatenation of the arguments.

Usage Examples See Example: Limiting the Number of E1 Interfaces, Example: Controlling IS-IS and MPLS Interfaces, Example: Adding T1 Interfaces to a RIP Group, Example: Configuring Administrative Groups for LSPs, and Example: Configuring Dual Routing Engines.

Related Documentation

- contains( ) on page 140
- starts-with( ) on page 142
- string-length( ) on page 142
- substring-after( ) on page 143
- substring-before( ) on page 143

contains()

Syntax *boolean* contains(*string*, *string*)

Description Return TRUE if the first string argument contains the second string argument, otherwise return FALSE.

Usage Examples See Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- concat() on page 139
- starts-with() on page 142
- string-length() on page 142
- substring-after() on page 143
- substring-before() on page 143

count()

Syntax *number* count(*node-set*)

Description Return the number of nodes in the argument node-set.

Usage Examples See Example: Limiting the Number of E1 Interfaces.

Related Documentation

- last() on page 140
- name() on page 141
- not() on page 141
- position() on page 141

last()

Syntax *number* last()

Description Return the index of the last node in the list that is currently being evaluated, which is equal to the number of items in the processed node list.

Usage Examples See Example: Limiting the Number of E1 Interfaces.

Related Documentation

- count() on page 140
- name() on page 141
- not() on page 141
- position() on page 141

name()

Syntax *string* name(*<node-set>*)

Description Return the full name of the first node in the node set, including the prefix for its namespace declared in the source document. If no argument is passed, the function returns the full name of the context node.

Usage Examples See jcs:emit-change Template.

Related Documentation
- count() on page 140
- last() on page 140
- not() on page 141
- position() on page 141

not()

Syntax *boolean* not(*boolean*)

Description Return TRUE if the argument is FALSE, and FALSE if the argument is TRUE.

Usage Examples See Example: Requiring and Restricting Configuration Statements, Example: Controlling IS-IS and MPLS Interfaces, Example: Configuring a Default Encapsulation Type, Example: Controlling LDP Configuration, Example: Adding a Final then accept Term to a Firewall, Example: Configuring Administrative Groups for LSPs, Example: Configuring Dual Routing Engines, and Example: Preventing Import of the Full Routing Table.

Related Documentation
- count() on page 140
- last() on page 140
- name() on page 141
- position() on page 141

position()

Syntax *number* position()

Description Return the position of the context node among the list of nodes that are currently being evaluated.

Usage Examples See Example: Adding a Final then accept Term to a Firewall and Example: Prepending a Global Policy.

Related Documentation
- count( ) on page 140
- last( ) on page 140
- name( ) on page 141
- not( ) on page 141

starts-with( )

Syntax *boolean* starts-with(*string*, *string*)

Description Return TRUE if the first string argument starts with the second string argument, otherwise return FALSE.

Usage Examples See Example: Imposing a Minimum MTU Setting, Example: Limiting the Number of E1 Interfaces, Example: Limiting the Number of ATM Virtual Circuits, Example: Adding T1 Interfaces to a RIP Group, Example: Configuring a Default Encapsulation Type, and Example: Configuring Dual Routing Engines.

Related Documentation
- concat( ) on page 139
- contains( ) on page 140
- string-length( ) on page 142
- substring-after( ) on page 143
- string-length( ) on page 142
- substring-after( ) on page 143
- substring-before( ) on page 143

string-length( )

Syntax *number* string-length(<*string*>)

Description Return the number of characters in the string. If the argument is omitted, it returns the string value of the context node.

Usage Examples See Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation
- concat( ) on page 139
- contains( ) on page 140
- starts-with( ) on page 142
- substring-after( ) on page 143
- substring-before( ) on page 143

substring-after()

Syntax *string* substring-after(*string*, *string*)

Description Return the portion of the first string argument that follows the occurrence of the second argument substring within the first. If the second string is not contained in the first string, or if the second string is empty, the function returns an empty string.

Usage Examples See Example: Limiting the Number of E1 Interfaces and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- concat() on page 139
- contains() on page 140
- starts-with() on page 142
- string-length() on page 142
- substring-before() on page 143

substring-before()

Syntax *string* substring-before(*string*, *string*)

Description Return the portion of the first string argument that precedes the occurrence of the second argument substring within the first. If the second string is not contained in the first string, or if the second string is empty, the function returns an empty string.

Usage Examples See Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- concat() on page 139
- contains() on page 140
- starts-with() on page 142
- string-length() on page 142
- substring-after() on page 143

CHAPTER 12

Standard XSLT Elements and Attributes Used in Automation Scripts

xsl:apply-templates

Syntax

```
<xsl:apply-templates select="node-set-expression">
  <xsl:with-param name="qualified-name" select="expression">
    ...
  </xsl:with-param>
</xsl:apply-templates>
```

Description Apply one or more templates, according to the value of the **select** attribute. If the **select** attribute is not included, the script recursively processes all child nodes of the current node. If the **select** attribute is present, the processor only applies templates to the child elements that match the expression of the **select** attribute, which must evaluate to a node-set. The **<xsl:template>** instruction dictates which elements are transformed according to which template. The templates that are applied are passed the parameters specified by the **<xsl:with-param>** elements within the **<xsl:apply-templates>** instruction.

Attributes **select**—(Optional) Selects the nodes to which the processor applies templates. By default, the processor applies templates to the child nodes of the current node.

Usage Examples See Example: Adding a Final then accept Term to a Firewall and Example: Preventing Import of the Full Routing Table.

Related Documentation

- XSLT Templates Overview on page 58
- xsl:call-template on page 146
- xsl:param on page 152
- xsl:template on page 154
- xsl:variable on page 156
- xsl:with-param on page 157

xsl:call-template

Syntax

```
<xsl:call-template name="qualified-name">
  <xsl:with-param name="qualified-name" select="expression">
    ...
  </xsl:with-param>
</xsl:call-template>
```

Description Call a named template. The **<xsl:with-param>** elements within the **<xsl:call-template>** instruction define the parameters that are passed to the template.

Attributes **name**—Specifies the name of the template to call.

Usage Examples See Example: Requiring and Restricting Configuration Statements, Example: Imposing a Minimum MTU Setting, and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- XSLT Templates Overview on page 58
- xsl:apply-templates on page 146
- xsl:param on page 152
- xsl:template on page 154
- xsl:variable on page 156
- xsl:with-param on page 157

xsl:choose

Syntax

```
<xsl:choose>
  <xsl:when test="boolean-expression">
  ...
  </xsl:when>
  <xsl:otherwise>
  ...
  </xsl:otherwise>
</xsl:call-template>
```

Description

Evaluate multiple conditional tests, and execute instructions for the first test that evaluates to TRUE or execute an optional default set of instructions if all tests evaluate to FALSE. The **<xsl:choose>** instruction contains one or more **<xsl:when>** elements, each of which tests a Boolean expression. If the test evaluates to TRUE, the XSLT processor executes the instructions in the **<xsl:when>** element, and ignores all subsequent **<xsl:when>** elements. The XSLT processor processes only the instructions contained in the first **<xsl:when>** element whose **test** attribute evaluates to TRUE. If none of the **<xsl:when>** elements' **test** attributes evaluate to TRUE, the content of the optional **<xsl:otherwise>** element, if one is present, is processed.

Usage Examples

See Example: Configuring Dual Routing Engines, Example: Preventing Import of the Full Routing Table, and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- XSLT Programming Instructions Overview on page 64
- xsl:for-each on page 149
- xsl:if on page 150
- xsl:otherwise on page 151
- xsl:when on page 157

xsl:comment

Syntax

```
<xsl:comment>
  ...
</xsl:comment>
```

Description Generate a comment node within the final document. The content within the **<xsl:comment>** element determines the value of the comment. The content must not contain two hyphens next to each other (- -); this sequence is not allowed in comments.

XSLT files can contain ordinary comments delimited by **<!--** and **-->** such as **<!-- ... Insert your comment here ... -->**, but these are ignored by the processor. To generate a comment within the final document, use an **<xsl:comment>** element.

Usage Examples See Example: Adding a Final then accept Term to a Firewall.

Related Documentation

- xsl:import on page 150
- xsl:stylesheet on page 153

xsl:copy-of

Syntax

```
<xsl:copy-of select="expression"/>
```

Description Create a copy of what is selected by the expression defined in the **select** attribute. Namespace nodes, child nodes, and attributes of the current node are automatically copied as well.

Attributes **select**—Specifies an expression to select the nodes to be copied.

Usage Examples See Example: Requiring and Restricting Configuration Statements.

Related Documentation

- xsl:element on page 149
- xsl:text on page 155
- xsl:value-of on page 155

xsl:element

Syntax

```
<xsl:element name="expression"/>
```

Description Create an element node in the output document.

Attributes **name**—Specifies the name of the element to be created. The value of the **name** attribute can be set to an expression that is extracted from the input XML document and evaluated at run time. To do this, enclose an XML element in curly brackets (**{ }**), as in **<xsl:element name="{$isis-level-1}"**.

Usage Examples See Example: Creating a Complex Configuration Based on a Simple Interface Configuration.

Related Documentation

- xsl:copy-of on page 148
- xsl:text on page 155
- xsl:value-of on page 155

xsl:for-each

Syntax

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

Description Include a looping mechanism that repeats XSL processing for each XML element in the specified node-set. The element nodes are selected by the XPath expression defined by the **select** attribute. Each of the nodes is then processed according to the instructions contained in the **<xsl:for-each>** element.

Attributes **select**—Specifies an XPath expression that selects the nodes to be processed.

Usage Examples See Example: Requiring and Restricting Configuration Statements, Example: Imposing a Minimum MTU Setting, Example: Limiting the Number of E1 Interfaces, Example: Adding T1 Interfaces to a RIP Group, Example: Configuring Administrative Groups for LSPs, and Example: Configuring Dual Routing Engines.

Related Documentation

- XSLT Programming Instructions Overview on page 64
- XPath Overview on page 55
- xsl:choose on page 147
- xsl:if on page 150
- xsl:otherwise on page 151
- xsl:when on page 157

xsl:if

Syntax

```
<xsl:if test="expression">
  ...
</xsl:if>
```

Description Include a conditional construct that causes instructions to be processed if the expression held in the **test** attribute evaluates to TRUE.

Attributes **test**—Specifies the expression to evaluate.

Usage Examples See Example: Requiring and Restricting Configuration Statements, Example: Limiting the Number of E1 Interfaces, Example: Adding T1 Interfaces to a RIP Group, and Example: Configuring Dual Routing Engines.

Related Documentation

- XSLT Programming Instructions Overview on page 64
- xsl:choose on page 147
- xsl:for-each on page 149
- xsl:otherwise on page 151
- xsl:when on page 157

xsl:import

Syntax

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

Description Import rules from an external style sheet. Provides access to all the declarations and templates within the imported style sheet, and allows you to override them with your own if needed. Any **<xsl:import>** elements must be the first elements within the style sheet, the first children of the **<xsl:stylesheet>** document element. The path can be any URI. The **../import/junos.xsl** path shown in the syntax is standard for all commit scripts, op scripts, and event scripts.

Imported rules are overwritten by any subsequent matching rules within the importing style sheet. If more than one style sheet is imported, the style sheets imported last override each previous import where the rules match.

Attributes **href**—Specifies the location of the imported style sheet.

Usage Examples See Example: Adding a Final then accept Term to a Firewall, Example: Configuring a Default Encapsulation Type, Example: Configuring Dual Routing Engines, Example: Controlling IS-IS and MPLS Interfaces, Example: Prepending a Global Policy, and Example: Preventing Import of the Full Routing Table.

Related Documentation

- Junos Script Automation: Named Templates in the jcs Namespace Overview on page 32

- Junos Script Automation: Global Parameters and Variables in the junos.xsl File on page 42
- Required Boilerplate for Commit Scripts
- Required Boilerplate for Event Scripts
- Required Boilerplate for Op Scripts
- xsl:stylesheet on page 153

xsl:otherwise

Syntax

```
<xsl:otherwise>
  ...
</xsl:otherwise>
```

Description Within an **<xsl:choose>** instruction, include a default set of instructions that are processed if none of the expressions defined in the **test** attributes of the **<xsl:when>** elements evaluate to TRUE.

Usage Examples See Example: Configuring Dual Routing Engines and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- XSLT Programming Instructions Overview on page 64
- xsl:choose on page 147
- xsl:for-each on page 149
- xsl:if on page 150
- xsl:when on page 157

xsl:param

Syntax

```
<xsl:param name="qualified-name" select="expression">
  ...
</xsl:param>
```

Description Declare a parameter for a template or for the style sheet as a whole. A template parameter must be declared within the template element. A global parameter, the scope of which is the entire style sheet, must be declared at the top level of the style sheet.

Attributes **name**—Defines the name of the parameter.

select—(Optional) XPath expression defining the default value for the parameter, which is used if the person or client application that executes the script does not explicitly provide a value. The **select** attribute or the content of the **<xsl:param>** element can define the default value. Do not specify both a **select** attribute and content; we recommend using the **select** attribute so as not to create a result tree fragment.

Usage Examples See Example: Requiring and Restricting Configuration Statements, Example: Imposing a Minimum MTU Setting, Example: Limiting the Number of E1 Interfaces, Example: Limiting the Number of ATM Virtual Circuits, and Example: Preventing Import of the Full Routing Table.

Related Documentation

- XSLT Parameters Overview on page 60
- XSLT Templates Overview on page 58
- **xsl:apply-templates on page 146**
- **xsl:call-template on page 146**
- **xsl:template on page 154**
- **xsl:variable on page 156**
- **xsl:with-param on page 157**

xsl:stylesheet

Syntax

```
<xsl:stylesheet version="1.0" xmlns:ext="URI">
  <xsl:import href="../import/junos.xsl"/>
  ...
</xsl:stylesheet>
```

Description

Include the document element for the style sheet. This element defines the root element of the style sheet, which contains all the top-level elements such as global variable and parameter declarations, import elements, and templates. Optionally, namespace mappings, which include an extension prefix and Uniform Resource Identifier (URI), can be included as attributes in the opening **<xsl:stylesheet>** tag.

Any **<xsl:import>** elements must be the first elements within the style sheet, the first children of the **<xsl:stylesheet>** document element. The path can be any Uniform Resource Identifier (URI). The **../import/junos.xsl** path shown in the syntax is standard for all commit scripts, op scripts, and event scripts.

Attributes

version—Specifies the version of XSLT that is being used. Junos OS supports XSLT version 1.0.

xmlns:*ext*="*URI*"—(Optional) Maps a namespace prefix to the URI for extension elements.

Usage Examples

See Example: Adding a Final then accept Term to a Firewall, Example: Configuring Administrative Groups for LSPs, Example: Configuring a Default Encapsulation Type, and Example: Customizing Output of the show interfaces terse Command Using an Op Script.

Related Documentation

- Required Boilerplate for Commit Scripts
- Required Boilerplate for Event Scripts
- Required Boilerplate for Op Scripts
- XSLT Namespace on page 55
- xsl:import on page 150

xsl:template

Syntax

```
<xsl:template match="pattern" mode="qualified-name" name="qualified-name
priority="integer">
  <xsl:param name="qualified-name" select="expression">
    ...
  </xsl:param>
  ...
</xsl:stylesheet>
```

Description

Declare a template that contains rules to apply when a specified node is matched. The **match** attribute associates the template with an XML element. The **match** attribute can also be used to define a template for a whole branch of an XML document. For example, **match="/"** matches the root element of the document. Although the **match** and **name** attributes are optional, one of the two attributes must be included in the template definition.

When templates are applied to a node set using the **<xsl:apply-templates>** instruction, they might be applied in a particular mode; the **mode** attribute in the **<xsl:template>** instruction indicates the mode in which a template needs to be applied for the template to be used. If templates are applied in the specified mode, the **match** attribute is used to determine whether the template can be used with the particular node. If more than one template matches a node in the specified mode, the priority attribute determines which template is used. The highest priority wins. If no priority is specified explicitly, the priority of a template is determined by the **match** attribute.

You can pass template parameters using the **<xsl:with-param>** element. To receive a parameter, the template must contain an **<xsl:param>** element that declares a parameter of that name. These parameters are listed before the body of the template, which is used to process the node and create a result.

Attributes

match—(Optional) XPath expression specifying the nodes to which to apply the template. If this attribute is omitted, the **name** attribute must be included.

mode—(Optional) Indicate the mode in which a template needs to be applied for the template to be used.

name—(Optional) Specify a name for the template. Named templates can be explicitly called with the **<xsl:call-template>** element. If the **name** attribute is omitted, the **match** attribute must be included.

priority—(Optional) Specify a numeric priority for the template.

Usage Examples

See Example: Adding a Final then accept Term to a Firewall, Example: Adding T1 Interfaces to a RIP Group, Example: Automatically Configuring Logical Interfaces and IP Addresses, Example: Requiring and Restricting Configuration Statements, and Example: Customizing Output of the show interfaces terse Command Using an Op Script.

Related Documentation

- XSLT Templates Overview on page 58

- XSLT Parameters Overview on page 60
- xsl:apply-templates on page 146
- xsl:call-template on page 146
- xsl:param on page 152
- xsl:variable on page 156
- xsl:with-param on page 157

xsl:text

Syntax

```
<xsl:text>
  ...
</xsl:text>
```

Description Insert literal text in the output.

Usage Examples See Example: Requiring and Restricting Configuration Statements, Example: Imposing a Minimum MTU Setting, Example: Limiting the Number of E1 Interfaces, Example: Controlling IS-IS and MPLS Interfaces, and Example: Adding a Final then accept Term to a Firewall.

xsl:value-of

Syntax

```
<xsl:value-of select="expression"/>
```

Description Extract the value of an XML element and insert it into the output. The **select** attribute specifies the XPath expression that is evaluated. In the XPath expression, use @ to access attributes of elements. Use " **.** " to access the contents of the element itself. If the result is a node set, the **<xsl:value-of>** instruction adds the string value of the first node in that node set; none of the structure of the node is preserved. To preserve the structure of the node, you must use the **<xsl:copy-of>** instruction instead.

Attributes **select**—XPath expression specifying the node or attribute to evaluate.

Usage Examples See Example: Imposing a Minimum MTU Setting, Example: Limiting the Number of E1 Interfaces, Example: Controlling IS-IS and MPLS Interfaces, Example: Configuring Administrative Groups for LSPs, and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- xsl:copy-of on page 148

xsl:variable

Syntax

```
<xsl:variable name="qualified-name" select="expression">
  ...
</xsl:variable>
```

Description Declare a local or global variable. If the **<xsl:variable>** instruction appears at the top level of the style sheet as a child of the **<xsl:stylesheet>** document element, it is a global variable with a scope that includes the entire style sheet. Otherwise, it is a local variable with a scope of its following siblings and their descendants.

Attributes **name**—Specifies the name of the variable. After declaration, the variable can be referred to within XPath expressions using this name, prefixed with the **$** character.

select—(Optional) Determines the value of the variable. The value of the variable is determined either by the **select** attribute or by the contents of the **<xsl:variable>** element. Do not specify both a **select** attribute and some content; we recommend using the **select** attribute so as not to create a result tree fragment.

Usage Examples See Example: Limiting the Number of E1 Interfaces, Example: Limiting the Number of ATM Virtual Circuits, Example: Configuring Administrative Groups for LSPs, and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- XSLT Variables Overview on page 63
- xsl:apply-templates on page 146
- xsl:call-template on page 146
- xsl:param on page 152
- xsl:template on page 154
- xsl:with-param on page 157

xsl:when

Syntax

```
<xsl:when test="boolean-expression">
  ...
</xsl:when>
```

Description Within an **<xsl:choose>** instruction, specify a set of processing instructions that are executed when the expression specified in the **test** attribute evaluates to TRUE. The XSLT processor processes only the instructions contained in the first **<xsl:when>** element whose **test** attribute evaluates to TRUE. If none of the **<xsl:when>** elements' **test** attributes evaluate to TRUE, the content of the **<xsl:otherwise>** element, if there is one, is processed.

Attributes **test**—Specifies a Boolean expression.

Usage Examples See Example: Configuring Dual Routing Engines, Example: Preventing Import of the Full Routing Table, and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- XSLT Programming Instructions Overview on page 64
- xsl:choose on page 147
- xsl:for-each on page 149
- xsl:if on page 150
- xsl:otherwise on page 151

xsl:with-param

Syntax

```
<xsl:with-param name="qualified-name" select="expression">
  ...
</xsl:with-param>
```

Description Specify a parameter to pass into a template. This element can be used when applying templates with the **<xsl:apply-templates>** instruction or when calling templates with the **<xsl:call-template>** instruction.

Attributes **name**—Specifies the name of the parameter.

select—(Optional) XPath expression specifying the value of the parameter. The value of the parameter is determined either by the **select** attribute or by the contents of the **<xsl:with-param>** element. Do not specify both a **select** attribute and content. We recommend using the **select** attribute to set the parameter so as to prevent the parameter from being passed a result tree fragment as its value.

Usage Examples See Example: Configuring Dual Routing Engines, Example: Preventing Import of the Full Routing Table, and Example: Automatically Configuring Logical Interfaces and IP Addresses.

Related Documentation

- XSLT Templates Overview on page 58
- xsl:apply-templates on page 146
- xsl:call-template on page 146
- xsl:param on page 152
- xsl:template on page 154
- xsl:variable on page 156

PART 3

Administration

CHAPTER 13

Operational Commands

request system scripts convert

Syntax **request system scripts convert (slax-to-xslt | xslt-to-slax) source *source/filename* destination *destination/<filename>***

Release Information Command introduced in Junos OS Release 8.2.
Command introduced in Junos OS Release 9.0 for EX Series switches.

Description Convert an Extensible Stylesheet Language Transformations (XSLT) script to Stylesheet Language, Alternative syntaX (SLAX), or convert a SLAX script to XSLT.

Options destination *destination/<filename>*—Specify a destination for the converted file.

Optionally, you can specify a filename for the converted file. If you do not specify a filename, the software assigns one automatically. The default destination filename is the same as the source filename, except the file extension is altered. For example, the software converts a source file called **test.xsl** to **test.slax**. The software converts a source file called **test1.slax** to **test1.xsl**.

slax-to-xslt—Convert a SLAX script to XSLT.

source *source/filename*—Specify a source file that you want to convert.

xslt-to-slax—Convert an XSLT script to SLAX.

Required Privilege Level maintenance

List of Sample Output
request system scripts convert slax-to-xslt on page 162
request system scripts convert xslt-to-slax on page 162

Output Fields When you enter this command, you are provided feedback on the status of your request.

Sample Output

request system scripts convert slax-to-xslt

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

request system scripts convert xslt-to-slax

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

PART 4

Index

Index

F

G

I

J

R

S

T

U

V

W

X