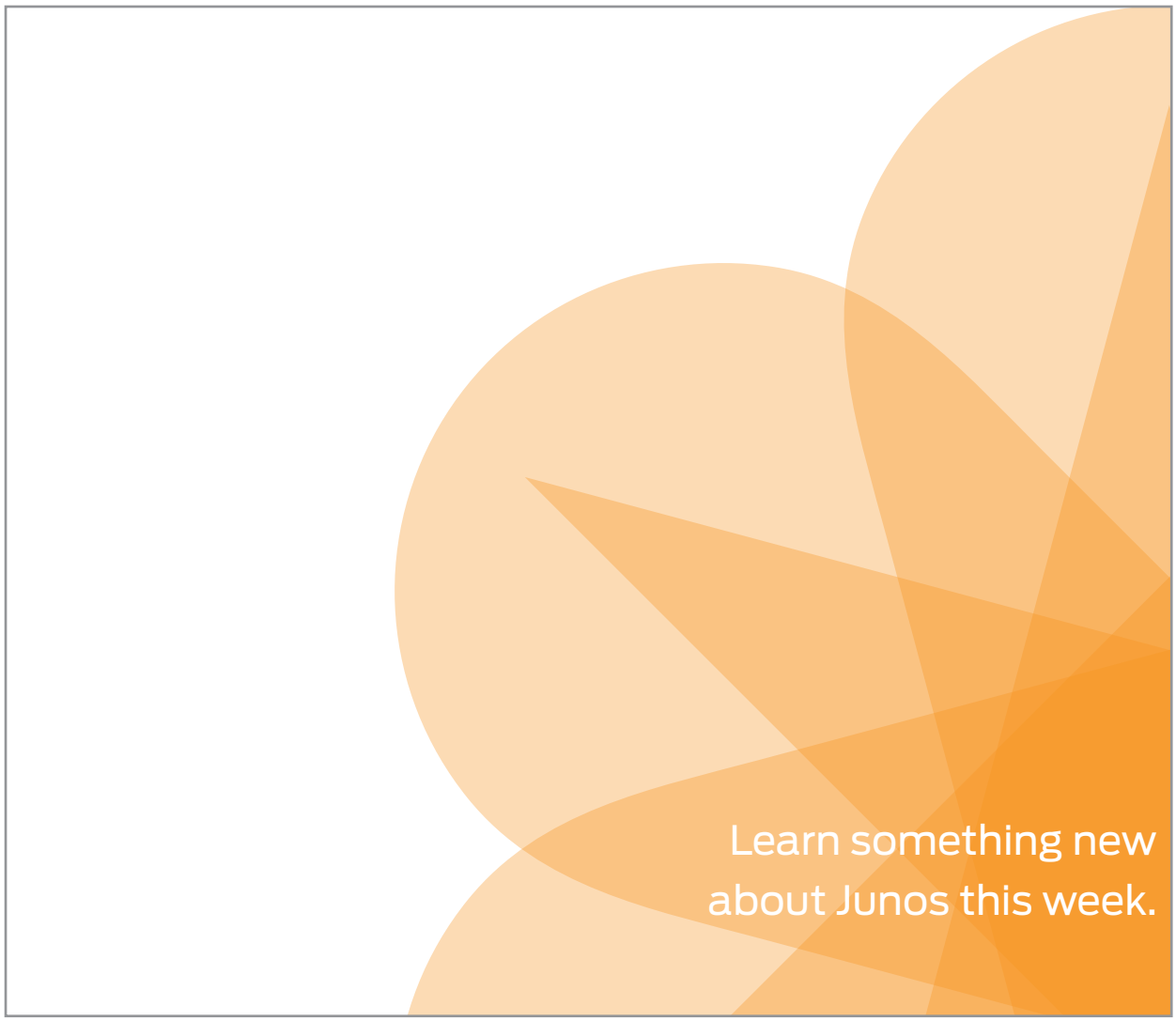
Junos® Automation Series

THIS WEEK:
APPLYING JUNOS AUTOMATION



By Curtis Call

THIS WEEK:
APPLYING JUNOS AUTOMATION

As you work with the Junos® operating system, you will build a knowledge reservoir of best practices and lessons learned, a body of intelligence that can be available 24x7 to help your network run optimally. Junos automation allows you to automate your accumulated intelligence through scripts which automatically control Junos devices according to your desired best practices. This book demonstrates how to implement this inherent potential in the Junos operating system.

Previously published as three separate *Day One* guides, *This Week: Applying Junos Automation* now combines Junos operation, event, and configuration automation techniques into a single, comprehensive volume.

"Junos automation technology provides a rich portfolio of toolsets that are extremely powerful yet simple to adopt. This book demonstrates that in very little time you too can create solutions for many challenging network management tasks."

Lixun Qi, Lead IP Engineer, T-Systems North America Inc.

"The flexibility and power of Junos configuration is increased with the introduction of commit scripts. This book provides a clear overview and the detailed information required to take full advantage of these scripts."

Mike Benjamin, Distinguished Engineer, Global Crossing

LEARN SOMETHING NEW ABOUT JUNOS THIS WEEK:

- Learn to use reference scripts from this book and Juniper's script library; Interpret the XML data structures used by Junos devices; communicate with Junos through the Junos XML API; ease how you write XML data structures using the SLAX XML abbreviated format; and, create your own customized operation scripts.
- Understand the difference between an op script and an event script; identify potential events that could be automated; build the needed event policy to match desired events and conditions; and, create your own customized event scripts.
- Understand the role of and possible uses for commit scripts; provide feedback as part of the commit process through warning or syslog messages; halt the commit process with error messages; alter the configuration through commit scripts; and, ceate your own customized commit scripts.

Published by Juniper Networks Books
www.juniper.net/books

ISBN 978-1936779161
52200
9 781936 779161

JUNIPER NETWORKS

7100140

Junos® Automation Series

This Week: Applying Junos Automation

By Curtis Call

JUNIPER® NETWORKS

Published by Juniper Networks Books
Editor in Chief: Patrick Ames
Copyeditor and Proofing: Nancy Koerbel
Junos Program Manager: Cathy Gadecki

About the Author
Curtis Call is a Systems Engineer at Juniper Networks. He is JNCIE-M #43 and has eight years experience working with Junos devices.

Author's Acknowledgmentsu
The author would like to thank all those who helped in the creation of this book. The literary manager, Patrick Ames worked with me to find the right outlet for this material and Nancy Koerbel fine-tuned my writing. The Day One Series Editor Cathy Gadecki was instrumental in bringing this project to fruition and helped me position the content to be more instructional. Roy Lee, the Junos automation Product Line Manager, reviewed the manuscript several times and always found ways to clarify the presentation. Thank you all.

NOTE: This book was first published as three separate *Day One* books in the Junos Automation Series.

This book is available in a variety of formats at: www.juniper.net/dayone.

Send your suggestions, comments, and critiques by email to: dayone@juniper.net.

Be sure to follow this and other Juniper Networks Books on: Twitter: @Day1Junos

Version History: v1 (This Week) February 2011

ISBN: 978-1-936779-16-1 (print)
Printed in the USA by Vervante Corporation.

ISBN: 978-1-936779-17-8 (ebook)

Juniper Networks Books are printed in the USA by Vervante Corporation and are available in bound editions at: www.vervante.com

2 3 4 5 6 7 8 9 10 #7100140-en

Welcome to *This Week*

This Week books are an outgrowth of the extremely popular *Day One* book series published by Juniper Networks Books. *Day One* books focus on providing just the right amount of information that you can do, or absorb, in a day. On the other hand, *This Week* books explore networking technologies and practices that in a classroom setting might take several days to absorb. Both book series are available from Juniper Networks at: www.juniper.net/dayone.

This Week is a simple premise – you want to make the most of your Juniper equipment, utilizing their features and connectivity – but you don't have time to search and collate all the expert-level documents on a specific topic. *This Week* books collate that information for you, and in about a week's time, you'll learn something significantly new about Junos that you can put to immediate use.

This Week books are written by Juniper Networks subject matter experts and are professionally edited and published by Juniper Networks Books. They are available in multiple formats, from eBooks to bound paper copies, so you can choose how you want to read and explore Junos, be it on the train or in front of terminal access to your networking devices.

What You Need to Know Before Reading

Before reading this book, you should be familiar with the basic administrative functions of the Junos operating system. This includes the ability to work with operational commands and to read, understand, and change the Junos configuration. Juniper's *Day One* books (www.juniper.net/dayone), as well as the training materials available on the Fast Track portal, can help to provide this background (see the last page of this book for these and other references).

Other things that you will find helpful as you explore the pages of this book:

- Having access to a Junos device while reading this book is very useful. A number of practice examples that reinforce the concepts being taught are included. Most of these examples require creating or modifying a script and then running it on a Junos device in order to see and understand the effect.
- The best way to edit SLAX scripts is to use a text editor on your local PC or laptop and then to transfer the edited file to the Junos device using a file transfer application. Doing this requires access to a basic ASCII text editor on your local computer as well as software to transfer the updated script using scp or ftp.
- While a programming background is not a prerequisite for using this book, a basic understanding of programming concepts is beneficial.

What This Book Can Do for You

This book helps you to automate operations tasks in your devices. It is divided into three parts:

Part One: Applying Junos Operations Automation

Read this part to learn how to use the Junos automation scripting toolset and how to write your first *operation* scripts. When you're done with this part, you'll be able to:

- Understand how the Junos automation tools work.
- Explain where to use the different Junos script types.
- Use reference scripts from this book and Juniper's script library.
- Interpret the XML data structures used by Junos devices.
- Communicate with Junos through the Junos XML API.
- Ease how you write XML data structures using the SLAX XML abbreviated format.
- Read SLAX scripts and understand the operations they perform.
- Create your own customized operation scripts.

Part Two: Applying Junos Event Automation

Part Two helps you to automate system events in your devices. Use this part to learn how to use the Junos automation scripting toolset and how to write your first *event* scripts. When you're done with this part, you'll be able to:

- Understand the difference between an op script and an event script.
- Identify potential events that could be automated.
- Build the needed event policy to match desired events and conditions.
- Correlate multiple events and determine the proper response to those events based on their relationship to each other.
- Create your own customized event scripts.

Part Three: Applying Junos Configuration Automation

Part Three helps you to automate the commit process of your Junos device. Read it to learn how to use the Junos automation scripting toolset and how to to write your first *commit* scripts. When you're done with this part, you'll be able to:

- Understand the role of and possible uses for commit scripts.
- Provide feedback as part of the commit process through warning or syslog messages.
- Halt the commit process with error messages.
- Alter the configuration through commit scripts.
- Use configuration macros to simplify your configuration or to store specialized data.
- Create your own customized commit scripts

Part One

Applying Junos Operations Automation

Chapter 1

Introducing Junos Automation

Computer networks continue to improve – promising higher speeds, more capabilities, and increased reliability. Yet enhanced functionality carries with it an increase in complexity, as more technologies have to coexist and work together. This tradeoff presents a challenge to network operators who want the advantages of new opportunities but still need to keep their networks as simple as possible in order to minimize operating costs and prevent errors.

Deploying Junos devices within a network can reduce the level of complexity that would otherwise be present. This benefit comes from the ability to use the same operating system to control routers, switches, and security devices. Instead of having to train staff to support multiple operating systems for each type of device, only a single operating system has to be learned and maintained. This decreases the overall complexity of the network.

As an organization continues to work with Junos it will build a knowledge reservoir of best practices and lessons learned. Imagine if this accumulated experience could always be available to help the network run optimally. Imagine if every configuration change, every system event, and every troubleshooting step could take advantage of the organization's gathered knowledge and make use of it. Enter Junos automation. It allows organizations to automate their pooled intelligence through scripts that automatically control Junos devices according to the desired best practices.

Junos automation is a standard part of the Junos operating system available on all Junos devices, including routers, switches, and security devices. This book introduces Junos automation and demonstrates how to take advantage of its potential. It also explains how to use operation scripts, one type of Junos automation script.

Junos automation enables an organization to embed its wealth of knowledge and experience of operations directly into its Junos devices:

- Business rules automation: compliance checks can be enforced. Change management can help to avert human error.
- Provisioning automation: complex configurations can be abstracted and simplified. Errors can be automatically corrected.
- Operations automation: customized commands and outputs can be created to streamline tasks and ease troubleshooting.
- Event automation: responses can be pre-defined for events allowing the device to monitor itself and react as desired.

What Junos Automation Can Do

Junos automation is a powerful suite of tools for automating the methods and procedures of operating a network. Automation can not only save your team time, it also helps to establish high performance operation of the network and to manage greater scale in the network by simplifying complex tasks.

The tool sets let you automate a majority of the commands used within the Junos command-line, further control the commit process, as well as automate the response to defined events. Junos includes three types of automation scripts, each providing different types of functionality for automation:

- Operation (op) scripts instruct Junos of actions to take whenever the script is called through the command-line or by another script.

- Event scripts instruct Junos of actions to take in response to an occurrence in a monitored set of events.
- Commit scripts instruct Junos of actions to take during the commit process of activating configuration changes.

MORE? To see examples of each type of script go to the online script library at www.juniper.net/scriptlibrary.

Operation (Op) Scripts

This book helps you to write your first op scripts. Op scripts are used in operational mode to create custom commands and to change configurations. They execute whenever called upon, either by an operator who simply enters a command request in the CLI or from within an event script (see below).

Tailor made show commands are the most common form of op scripts. An op script written for this objective gathers data from multiple show commands, processes it, and then outputs the desired information to the screen.

Another common form of op script is an automated configuration change. These op scripts perform controlled configuration changes based on supplied input from command line arguments, interactive prompts, or Junos show commands. The advantage of this approach is that you can code the structure of the change into the script itself. This mitigates human error and allows users with less expertise the ability to change the configuration in controlled ways.

You can also create op scripts to iteratively narrow the potential cause of network problems. These scripts run an operational mode command, process the output, determine the next appropriate action, and repeat the process until the source of the problem is determined and reported to the CLI. Op scripts can thereby give operators a running start that is immensely valuable during troubleshooting.

By uncovering the root cause, or at least helping operators to quickly shorten the list of possible causes, op scripts can speed the time to resolution. Rapid problem diagnosis is crucial during an outage; it is not uncommon for an operations team to spend hours diagnosing a problem that ultimately takes only a few minutes to repair.

Event Scripts

Event scripts are triggered automatically in response to an event (or events) such as a syslog message, SNMP trap, or timer. You can use event scripts to gather troubleshooting information in reaction to a system event, or to automatically change the configuration due to networking changes or the current time of day.

A typical event script anticipates a scenario such as: “If interface X and VPN Y go down, execute op script XYZ and log a customized message.” By watching for events specified by each organization—warning parameters that signal potential problems such as a line card crash, routing failure, or memory overload—operations teams can respond more quickly with corrective actions.

The event scripts work by processing new events. When the event process of the Junos operating system receives events, a set of event scripts can instruct it to select and correlate specific events. An event script can also perform a set of actions, such as invoking a Junos command or an op script, creating a log file from the command output, and uploading the file to a given destination.

In this way, operators can better control the series of operations events from the first leading indicators. Event scripts and op scripts work together to automate early warning systems that not only detect emerging problems, but can also take immediate steps to restore normal operations. By capturing more directly relevant information faster and taking action, automation scripts can give operations teams more options earlier.

MORE? After you learn about scripting basics in this book, download *Day One: Automating Junos Operations* to learn more about using op and event scripts. Check for availability at www.juniper.net/dayone/.

Commit Scripts

Commit scripts instruct Junos during the commit process of configuration changes. When the user performs a commit operation, Junos executes each commit script in turn, passing the candidate configuration through the defined commit scripts. This allows the script to fail the commit when user-defined errors are present, to provide warnings to the committing user on the console, to log messages to the syslog, or to change the configuration automatically. As examples, you could use a commit script to enforce physical card placement and wiring standards, or for a specified logical configuration.

The true power of commit scripts becomes evident when they are used as macros. Macros greatly simplify the task of complex configurations by taking basic configuration variables as input (such as the local interface, the VPN ID, and the Site ID), and then using these to create a complete set of configuration statements (such as a VPLS interface). By limiting user input only to necessary variables, macros can ensure consistency in the configuration of a particular interface, protocol, etc. across the network.

MORE? Download *Day One: Automating Junos Configuration* to learn more about using commit scripts. Check for availability at www.juniper.net/dayone.

How Junos Automation Works

Junos automation scripts provide a sequenced set of steps (or conditional steps) that Junos takes when it processes the script. Junos can process only those scripts specifically included as being a part of the device configuration. Only specifically permitted users have the permission to add a script to the device configuration.

Figure 1.1 outlines the basic flow of script processing. Junos stores scripts in predetermined `/var/db/scripts/` directories. Junos begins the processing of a script as a result of a trigger, defined by the type of script. For example, the trigger for processing all commit scripts is the `commit` command in configuration mode. A script engine within the the Junos operating system then processes the script line-by-line, taking the specified actions. These actions can include requests to other Junos processes. When the script engine completes the processing of the script, it sends the results to the output specified by the script.

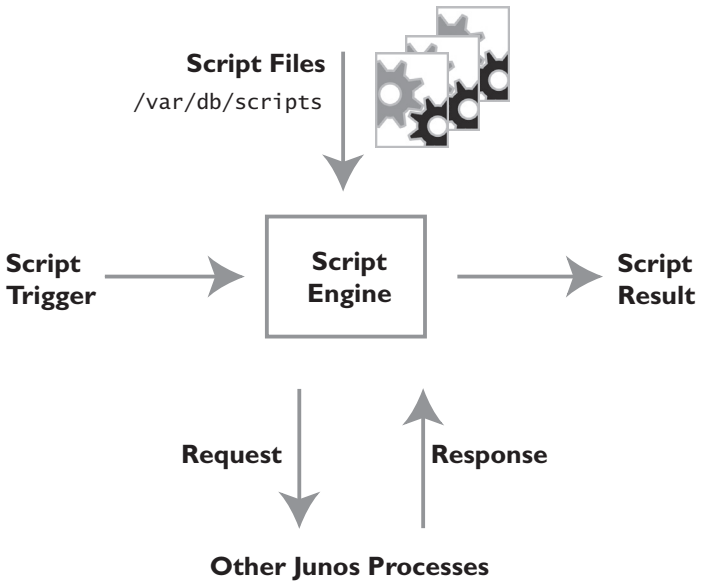


Figure 1.1 The Flow of Script Processing

NOTE The script engine uses XML (eXtensible Markup Language) to read the script and communicate with other Junos processes. The management process daemon, known as `mgd`, of the Junos operating system includes the primary script engine for processing scripts. The event process daemon, known as eventd, also includes a script engine for monitoring events. The Configuration and Diagnostic Automation Guide includes further details about how Junos processes scripts. Find the guide along with other Junos documentation at www.juniper.net/techpubs/.

Scripting Languages

Junos automation scripts can be written in either of two scripting languages: XSLT or SLAX. XSLT is a standardized language designed to convert one XML document into another. While XSLT can be used to write Junos automation scripts, its original purpose of document conversion and the fact that it's written in XML makes it a less comfortable choice for most people.

NOTE XSLT stands for eXtensible Stylesheet Language Transformations. SLAX stands for Stylesheet Language Alternative syntaX.

Juniper developers created SLAX to provide a more user-friendly and intuitive method in which to write Junos scripts than XSLT. SLAX has a more readable syntax. And, it feels more natural to anyone who is familiar with reading Junos configurations or writing programs in the C or Perl programming languages.

This book focuses solely on teaching the SLAX language, as XSLT knowledge is not necessary to take advantage of Junos automation.

MORE? For more on how to use XSLT to write Junos scripts, see the *Configuration and Diagnostic Automation Guide* at www.juniper.net/techpubs/.

Using Junos Automation with Other Systems

Junos automation complements existing network automation systems. Existing systems offer substantial benefits for change management, provisioning, and monitoring, but their usefulness is limited when it comes to detecting and diagnosing configuration and network problems.

Automated change management systems can only identify problems after the fact, as these packages collect information about system conditions reactively, by polling the device at predefined intervals. Junos automation is unique in that it provides immediate, on-box problem detection and resolution. The automation scripts are always available, always alert to potential issues, and always ready to initiate repair.

TIP Building on the Junos automation toolset, Juniper Networks Advanced Insight Solutions (AIS) introduces intelligent self-analysis capabilities directly into platforms running Junos. AIS provides a comprehensive set of tools and technologies designed to enable Juniper Networks Technical Services with the automated delivery of tailored, proactive network intelligence and support services. For more information visit the Juniper Networks services web page at http://www.juniper.net/us/en/products-services/technical-services/j-care/.

XML Basics

Junos automation scripts communicate with their host device using the XML language. While it's somewhat of a dry topic, a basic understanding of how XML is used in the Junos operating system is thereby a necessary first step in learning how to apply Junos scripts. This section gives you just the brief XML background that you need for writing your own scripts.

Fortunately, the SLAX language greatly simplifies how one reads and uses XML data structures. The next section explores how SLAX abstracts the described XML data structures for greater ease of use.

Displaying XML

The defined Junos XML API (Application Programming Interface) provides methods for Junos scripts to make requests. These requests can instruct other Junos processes to retrieve particular data or perform specific actions (see Figure 1.1). Junos performs the requested operation and returns the XML results to the script engine for further processing of the script.

As an example of the XML results that a Junos script can use, take a look at the configuration below expressed in XML mode:

```
user@Junos> show configuration routing-options | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.6I0/junos">
  <configuration junos:commit-seconds="1238100702" junos:commit-
localtime="2009-03-26 13:51:42 PDT" junos:commit-user="user">
     <routing-options>
       <route-distinguisher-id>192.168.1.1</route-distinguisher-id>
       <autonomous-system>
         <as-number>65535</as-number>
       </autonomous-system>
     </routing-options>
  </configuration>
  <cli>
```

```
    <banner></banner>
  </cli>
</rpc-reply>
```

At first glance this output can appear confusing, but the intuitive structure makes it simple to understand. Notice the `rpc-reply` mentioned in the first line of output. This shows the output is a reply from Junos providing the requested XML data.

The next line indicates that this is configuration information, and the following line begins the familiar `routing-options` configuration hierarchy. Just as in a normal configuration, the `routing-options` hierarchy contains the `route-distinguisher-id` and `autonomous-system` configurations. The XML form uses the same hierarchical approach, making it easy to understand and simple to compare against the text configuration.

The output above includes examples of key concepts necessary to understand how to communicate using the Junos XML API: elements, attributes, namespaces, and nodes.

Try It Yourself: Viewing Junos Configuration in XML

Show the following configuration hierarchy levels in XML on a Junos device:

```
(e.g. show configuration system | display xml)

[system]
[interfaces]
[protocols]
```

Elements

An element is the basic unit of information within an XML data structure. Elements can contain data such as a text string or number, or they can contain other elements. An element that contains another element is the parent of the enclosed child element. Likewise, the inner element is the child of the containing element. This creates a hierarchy, which is inherent in the XML structure, similar to the familiar Junos configuration hierarchy.

Elements are written by using start and end tags that provide the boundaries of the element. A tag contains the element name enclosed within < > arrows. The output above shows examples of tags, such as <routing-options>, a tag for the `routing-options` element. The output lists <routing-options> twice, once as the start tag and once as the end tag.

All the text within the start and end tags is an element's data. Both tags include the element name; however, the end tag also includes a / before the name, for example `</routing-options>`. If an element is empty, meaning it has no data or child elements, then it can be expressed using a single tag with a / following the element name, for example `<extensive/>`.

Here is an example XML configuration hierarchy showing four separate elements:

```
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
    <disable/>
  </interface>
</interfaces>
```

The <interfaces> element is the parent element of <interface>, which is the parent element of the <name> and <disable> elements. The <name> element contains the text data ge-0/0/0. The <disable> element, however, contains no data or child elements. This is why it is expressed as an empty tag instead of a start and end tag pair. Yet its presence in the XML data structure communicates that this interface has been disabled.

Attributes

Elements can include additional information in the form of attributes. This information is expressed by including the attribute name and value within the start tag:

```
user@Junos> show configuration routing-options | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.6I0/junos">
  <configuration junos:commit-seconds="1238100702" junos:commit-
localtime="2009-03-26 13:51:42 PDT" junos:commit-user="user">
<snip>
```

Both the <rpc-reply> and <configuration> elements have attributes defined. For example, the <configuration> element has three attributes: junos:commit-seconds, junos:commit-local-time, and junos:commit-user. Here, the three attributes of the <configuration> element provide additional details about the last commit.

XML expresses the attribute value by including an equal sign (=) following the attribute name and providing the value within quotation marks as shown above. If an element has multiple attributes, they are included within the start tag separated by spaces.

Namespaces

In the last example the three attributes defined for the <configuration> element all started with the same word. In this case the Junos portion of the attribute name fulfills a specific purpose: it indicates the namespace of the attribute.

SHORT CUT

A namespace prevents confusion between elements using the same name for different purposes. For example, there could be a commit-seconds attribute used by multiple computing devices, but when it is included in the Junos namespace it becomes junos:commit-seconds. What the attribute indicates is now explicitly known.

More precisely, the Junos name is actually a placeholder for the full namespace, which is http://xml.juniper.net/junos/9.6I0/junos. XML uses a URL for namespaces to ensure each is unique and to prevent namespace collisions. Fortunately XML and SLAX include ways to simplify the assignment of URLs to namespaces.

Defining a Namespace

Writing out the full URL-based namespace for every attribute or element can become overly verbose and tedious. For this reason XML enables the creation of a placeholder:

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.6I0/junos">
```

The single attribute of <rpc-reply> fulfills a special purpose. xmlns defines a XML namespace. The example declares that the http://xml.juniper.net/junos/9.6I0/junos namespace is the reference of Junos. This assignment takes effect for the <rpc-reply> element and all of its child elements.

Using Namespaces in SLAX

Junos further simplifies the use of namespaces when working with SLAX scripts. Rather than using the exact Junos version (9.6 in the above example), Junos replaces the version number with a * when providing the XML data structure to the script. In this way a script can be written without reference to the exact namespace used on the Junos device.

With this simplification, the only steps that a SLAX script writer must follow to use namespaces correctly are:

1. Copy the standard boilerplate (explained in Chapter 2) into the script.

2. Prepend the namespace placeholder (`junos, jcs, xnm`, etc.) correctly to the element or attribute name (if they have been assigned a namespace).

Nodes

When SLAX parses a XML data structure it reads it as a tree of nodes. Every element, attribute, and text data becomes a separate node on the tree. As an example, Figure 1.2 shows how SLAX would assemble all the nodes for the following configuration:

```
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
    <disable/>
  </interface>
</interfaces>
```

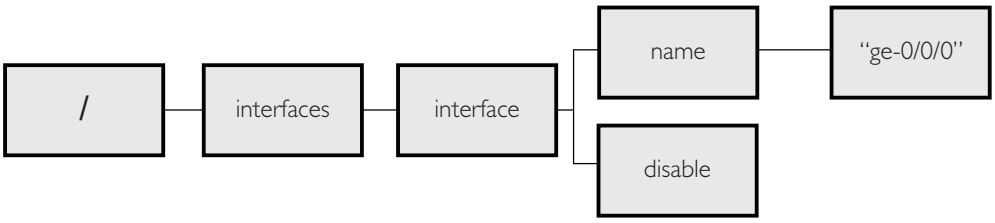


Figure 1.2 SLAX Tree of Nodes Example

Note that each node in Figure 1.2 is in the correct hierarchical position of the tree. In SLAX, every node tree contains a root node at its base, representing data to computers, with its syntax expressed in Figure 1.2 as `/`. Next the four element nodes appear according to their hierarchy. Lastly, a text node holds the text contents of the `<name>` element. With the XML data structure expressed in tree form, it is possible to traverse the tree from parent to child, and from sibling to sibling, in order to retrieve the necessary data.

MORE? For more details on XML you can look at http://www.w3schools.com/xml/. It is one of many online XML tutorials.

SLAX Abbreviated XML Format

Compare this XML data structure:

```
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
    <disable/>
```

```
  </interface>
</interfaces>
```

To the actual configuration it represents:

```
interfaces {
   ge-0/0/0 {
      disable;
   }
}
```

The XML data structure is harder to read and more time-consuming to write. While XML's structure makes it very consistent and useful for representing data, its syntax is not ideal to read or manually enter. This stems from the necessity of using start and end tags for each element.

The SLAX language therefore uses an abbreviated format to describe XML data structures. This format is more congruent with the Junos configuration style:

```
<interfaces> {
  <interface> {
    <name> "ge-0/0/0";
    <disable>;
  }
}
```

This is the same XML data structure as shown in the XML format example, yet in SLAX it appears more similar to the actual configuration it represents. Note one difference between the actual configuration text and its representation in the SLAX abbreviated XML format: the identifier for configuration objects appears as the first child element within an element called `<name>`.

As an example, ge-0/0/0 is assigned to the `<name>` child element of the `<interface>` element in the SLAX format.

To achieve the simplification, the SLAX abbreviated XML format uses only the start tags; the end tags are no longer required. Instead, SLAX expresses the boundary of the element in one of three ways:

- If the element contains child elements then curly braces `{ }` contain the child elements (the same method used to indicate hierarchy in Junos).
- If the element contains data then the data is written within quotation marks (`" "`) and the line is terminated with a semi-colon (`;`) (similar to Junos configurations).
- A single start tag terminated by a semi-colon (`;`) represents empty elements with no children or text data.

Try It Yourself: Writing XML in the SLAX Abbreviated Format

Rewrite the following configuration using the SLAX abbreviated XML format:

```
system {
  host-name r1;
  login {
    message "Unauthorized access prohibited.";
  }
}
```

Chapter 2

Writing Your First Script

Junos automation scripts can automate many operation steps in Junos. This chapter provides the first glimpse of how to write a script, load it on a Junos device, and enable it in the configuration.

Hello World

The functionality of this first op script is very simple: when run, it displays the text "Hello World!" on the console. To run the op script, an administrator simply enters the script file name at the CLI prompt. The complete code for the Hello World script follows:

```
version 1.0;

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

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

match / {
  <op-script-results> {
    <output> "Hello World!";
  }
}
```

Here is the output shown by the Hello World op script:

```
user@Junos> op hello-world
Hello World!
```

How to load and run the Hello World op script:

To run this op script on a device, take the following steps.

1. Save the code into a text file called hello-world.slax.

ALERT! All SLAX script filenames must end with the .slax extension.

2. Copy the script file into the /var/db/scripts/op directory on the Junos device.

3. Before you can run the script, you must enable it within the Junos configuration. Explicit configuration is a security precaution that prevents unauthorized scripts from being executed on the Junos device. Only super-users, users with the all permission bit, or users that have specifically been given the maintenance permission bit are permitted to enable or disable Junos scripts in the configuration. The command to enable an op script is set system scripts op file <filename>. So for the Hello World op script, enter:

```
set system scripts op file hello-world.slax
```

NOTE Devices with multiple routing-engines must have the script file copied into the /var/db/scripts/op directory on all routing-engines. The script must be enabled within the configuration of each routing-engine as well. Typically this configuration is done automatically through configuration synchronization. However, if the configurations are not synched, then the configuration must be entered manually into all routing-engines.

4. Execute the script with the op command followed by the script file name (without the .slax filename extension). For example:

```
user@Junos> op hello-world
```

SLAX Syntax Rules

The SLAX scripting language has a set of basic syntax rules. Chapter 1 provided some of these rules in the section on SLAX abbreviated XML format. Since Junos scripts contain XML elements and data structures, scripts must follow these relevant formatting rules.

The rest of the SLAX syntax rules are very similar to the Junos configuration syntax rules. For example, code blocks and line termination within SLAX scripts are done in the same manner as in Junos configuration, and the formatting of strings and comments in SLAX is also comparable to Junos.

Code Blocks

A Junos configuration indicates hierarchy through the use of curly braces { }:

```
interfaces {
  interface ge-0/0/0 {
    disable;
  }
}
```

In the above example, the interfaces configuration hierarchy contains the interface ge-0/0/0 hierarchy, which contains disable. The entire hierarchical relationship is clearly defined with the use of curly braces.

The SLAX scripting language follows a similar style, enclosing distinct code blocks within curly braces to indicate their hierarchy and bounds. Review this portion of the configuration from the Hello World script:

```
match / {
  <op-script-results> {
    <output> "Hello World!";
  }
}
```

Curly braces bound the match / code block. This provides a clear boundary indicating exactly where the code block starts, where it stops, and what code it contains.

Line Termination

The following lines end with a semi-colon:

```
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";
```

Each of these is an example of an individual statement. Individual statements are not part of a code-block. SLAX terminates individual statements with a semi-colon (the same as in Junos configuration). The semi-colon tells the script engine in Junos that the end of the line has been reached.

String Values

A string is a sequence of text characters. `"Hello World!"` and `"../import/junos.xsl"` are examples of strings in the Hello World script. Scripts must always enclose

string values within quotes. In this way the script engine knows that the text is intended as a string value.

This method is very similar to how Junos handles strings in a configuration. The difference: in a Junos configuration, quotation marks are generally only required when the text includes a space; while in a SLAX script, quotation marks are always required whether a space is present or not.

NOTE SLAX allows the use of either single quotes 'String Value' or double quotes "String Value", but the character used to open the string must also be used to close it.

Adding Comments

The regular use of comments within a script is very helpful and highly recommended. Comments provide insight into the logic of the script and the expectations it is working under. This can be beneficial to those that did not write the script and are unfamiliar with the design decisions that influenced it. Comments can also be useful for the script author who might need to revise the script months later.

Comments in SLAX scripts are written within the delimiters `/*` and `*/` such as:

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

This syntax is similar to how comments appear in a Junos configuration when configuration commands are annotated.

NOTE In Junos comments are indicated in two ways, either within /* and */ or following a #. SLAX scripts only support the delimiters /* and */.

Comments can be included anywhere within your script:

```
match / {
  <op-script-results> {
    /* Display this string on the console */
    <output> "Hello World!";
  }
}
```

Create multi-line comments by entering the terminating */ on a separate line from the starting /*:

```
/*
 * This is a simple script which outputs "Hello World!" to the console.
 */
```

Try It Yourself: Adding Comments to the Hello World Script

Make the following modifications to the Hello World script:

1. Add a multi-line comment at the beginning that describes the purpose of the script.

2. Add an additional comment before the <output> "`Hello World!`"; line to state that it is writing to the console.

After making the two modifications, replace the prior version of hello-world.slax on your Junos device with the new version. Execute the script again and verify that the new comments did not change the operation.

Understanding the Result Tree

Junos automation requires a communication method so that scripts can instruct Junos to perform desired actions. For example, the Hello World script causes Junos to display "Hello World!" on the console. In the Hello World script, the <output> element within the result tree provides this request.

Using the result tree is the simplest way for a script to provide instructions for Junos. The result tree is a XML data structure created by the processing of the script and delivered to the script engine after the script terminates. During operation, the script specifies the XML elements to include in the result tree. Once the script has finished, Junos follows the instructions of the completed result tree.

Writing to the Result Tree

Writing to the result tree within a script is easy. XML elements are simply embedded within the SLAX script in the abbreviated XML format. When the script engine arrives at a line that consists of a XML element it automatically attaches that element into the result tree.

NOTE Some XML elements take effect within the script itself and are not written into the result tree. These are special purpose elements such as <xsl:sort> and <xsl:message>. They constitute the exception to the rule; typically all embedded XML elements are written to the result tree.

Processing these lines creates the result tree in the Hello World script:

```
match / {
  <op-script-results> {
    <output> "Hello World!";
  }
}
```

<op-script-results> is an XML element with a child element of <output>. When the script engine begins executing the script it reaches this XML data structure, recognizes these as XML elements, and writes them to the result tree as:

```
<op-script-results> {
  <output> "Hello World!";
}
```

The parent element in the example above is <op-script-results>. This is always the top-level element in an op script result tree. This element indicates to Junos that instructions are coming from an op script. There is no action performed by the <op-script-results> element, it simply contains the child elements. It is the child elements that provide instructions for Junos to process.

The <output> element is the most common element found in the result tree of op scripts. As the name implies, it outputs an associated string. Specifically, it instructs Junos to display the string to the console followed by a line-feed. A script can include multiple <output> elements, with each string appearing on a different line:

```
<op-script-results> {
  <output> "Hello World!";
  <output> "I'm Home!";
}
```

Results in the following output:

```
Hello World!
I'm Home!
```

NOTE If you wish to include line-feeds within your text string then use the `\n` escape character: `<output> "First Line\nSecond Line";`

Try It Yourself: Adding Additional Output to the Hello World Script

Modify the Hello World script by adding two additional lines of output to the console above the "Hello World!" string.

Replace the prior version of hello-world.slax on your Junos device with the changed version. Execute the script again and see the effect the new <output> elements have on the script output.

Importing Script Code

The Hello World script example includes the following line:

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

This specific statement loads all the code from the /var/db/scripts/import/junos.xsl script file into your op script prior to execution. Importing allows the use of common code within multiple scripts without having to copy and paste the actual text from one script to the other. The junos.xsl script file is included as part of the standard Junos distribution. It contains default templates and parameters. Your scripts should always include the above line to import this file.

MORE? For information on the contents of junos.xsl see *The Configuration and Diagnostic Automation Guide* at www.juniper.net/techpubs/.

The Main Template

When writing a script, all code and result tree elements must be included within a code structure known as a template. When the script engine in Junos first executes a script, it searches the script file for the main template. The script engine then begins executing the instructions and writing the included result tree elements. For op scripts, the main template is match /.

Note in the Hello World script below: the presence of the match / template, the curly braces { } enclosing the code block, and the XML elements to add to the result tree included within the block.

```
match / {
  <op-script-results> {
    <output> "Hello World!";
  }
}
```

Using the Op Script Boilerplate

When writing Junos op scripts, work from the standard boilerplate:

```
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> {

  /* Your script code goes here */

  }
}
```

The boilerplate simplifies script writing by providing the needed name-space URLs (see Chapter 1) along with other components. Copy and paste the boilerplate and add your script code within it. The boilerplate includes the following components:

- version: while version 1.0 is currently the only available version of the SLAX language, the version line is required at the beginning of all Junos scripts.
- ns: a `ns` statement defines a namespace prefix and its associated namespace URL. The following three namespaces must be included in all Junos scripts:

```
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";
```

It is easiest to just copy and paste these namespaces into each new script as part of the boilerplate rather than trying to type them out by hand:

- import: the import statement is used to import code from one script into the current script. As the junos.xsl script contains useful default templates and parameters, all scripts should import this file. The import "../import/junos.xsl"; line from the boilerplate is all a script needs to accomplish this.
- match /: this code block is the main template of the op script. In the standard boilerplate it includes the <op-script-results> result tree element.

The boilerplate includes the `<op-script-results>` element to simplify writing of op scripts. SLAX statements can be included within the `<op-script-results>` code block without interfering with the created result tree. The script engine can differentiate between statements to execute and XML elements to add to the tree.

Try It Yourself: Writing Your Own Script Using the Boilerplate

Using the configuration boilerplate, create a new op script that outputs three separate lines of text to the console. Copy this script to your Junos device and enable it. Now you can verify it by executing it from the command-line.

Chapter 3

Understanding SLAX Language Fundamentals

Chapter 2 covered the syntax rules of the SLAX scripting language as well as the boilerplate used when creating a new op script. It also explored the Hello World op script and demonstrated how to write text to the console. This chapter digs deeper into the fundamentals of the SLAX language and further explains its capabilities.

- The `jcs`, `xnm`, and `junos` namespaces are reserved. Do not use any of these namespaces when creating variables, parameters, elements, or templates.
- Do not start any names with `"junos"`.

While the following serve as guidelines, they are also best practices that let your scripts conform to Junos configuration naming standards as well as to official Junos scripts.

- Write variables, parameters, elements, and templates entirely in lowercase.
- Separate multiple words with a dash (-), for example: `gig-interface`.

Variables

In the SLAX language, a variable is a reference to an assigned value. The variable name is used within the script, and the script engine substitutes the value in its place when it executes the script. SLAX variables are immutable; they cannot be changed. This might seem strange to those who are accustomed to other programming languages, but in SLAX variables always refer to the value to which they were first assigned.

Data Types

There are five data types defined in the SLAX scripting language:

- string: a sequence of text characters, for example "`Hello World!`".
- number: numbers are stored as floating points so decimals are permitted.
- boolean: used for conditional operations; evaluated as either true or false.
- result tree fragment: a portion of the result tree. By default, all XML elements that are embedded in a script are written to the result tree. It is possible, however, to redirect this XML data to a variable instead. The variable stores the data as an unparsed XML data structure, as such no additional data can be extracted from it. The script can only use the unparsed data to write to the result tree later or to convert to a node-set or string.
- node-set: an unordered set of XML nodes. A node-set consists of parsed XML data, so information can be extracted from it. Typically, node-sets are the result of a query to Junos for information, a location path, or a converted result tree fragment.

Declaring Variables

Variables are all declared using the `var` statement. Variable names are always preceded by a dollar sign:

```
var $example-string = "Example";
```

The data type of the variable is automatically determined based on the assigned value. Here are examples of how to declare variables in each of the different data types:

- string: `var $example-string = "Example";`
- number: `var $example-number = 15;`
- boolean: `var $example-boolean = ( 15 == 15 );`
- result tree fragment:
  ```
  var $example-rtf = {
   <system> {
     <host-name> "R1";
    }
   }
  ```
- node-set: `var $example-node-set = jcs:invoke("get-interface-information");`

BEST PRACTICE To allow your scripts to be compatible with future Junos script functionality, always follow these rules when naming variables, parameters, elements, and templates.

Using Variables

Once declared, a script can use the variable to reference the represented value. When using variables their full case-sensitive name must be used, including the preceding dollar sign:

```
match / {
  <op-script-results> {
    var $router-name = "R1";
    <output> $router-name;
  }
}
```

The above example shows the main template of an op script. This script declares a variable of `$router-name` with a string value of "`R1`" assigned to it. The script then includes this variable as the content of the `<output>` result tree element that writes "`R1`" to the console.

Scope of Variables

Variables are only usable within a limited scope. A scope is the code hierarchy in which a variable is declared. Each variable can be used within its own scope, as well as in other specific scopes that also fall within the declared scope.

The following are the main types of scopes for variables:

- global variable: refers to any variable declared outside of all templates. Global variables can be referenced anywhere within the script.
- template variable: refers to variables defined within templates, such as the main template, and have a scope of only their own template. Template variables cannot be used outside of their own template.

NOTE The script code can declare variables of the same name both globally and within a template, but only one or the other is usable at a time. The template variable overrides the global variable within the template that assigns the template variable.

More specific scopes are also possible. If a variable is declared within the code block of either an `if` or `for-each` statement (which are discussed later), then it is only usable within that code block; it cannot be referenced outside of it.

Global Variables

Here is a variation of the Hello World script where the string is defined as a global variable:

```
/* hello-world.slax */
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";

/* This is a Global Variable */
var $first-string = "Hello World!";

match / {
  <op-script-results> {

    /* This is a variable with template scope */
    var $second-string = "Goodbye World!";

  /* Output both variables to the console */
    <output> $first-string;
    <output> $second-string;
  }
}
```

In the above example, both the global variable `$first-string` and the template variable (sometimes called a local variable) `$second-string` are available for use within the main template. However, if additional templates are added to the script (as is discussed later in this chapter), only the global variable `$first-string` can be used within these.

NOTE Notice that the `var` statement is declared within the `<op-script-results>` XML element. The SLAX processor is able to determine that this is a line of script code rather than an XML element, and it does not try to write it to the result tree. It is common to interleave SLAX code and result tree elements in this manner within Junos scripts.

Operators

SLAX contains a wide variety of operators to enhance script operation. These operators enable the script to perform mathematical operations, compare values, convert data, and create complex expressions. Table 3.1 summarizes the operators available in SLAX.

Table 3.1 SLAX Operators

Name ... Code	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 and 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> $bit-count div 8;` Divides the bits by eight, returning the byte count, and displays the result on the console (requires that the variable `$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 ==	`$mtu == 1500` If the value assigned to `$mtu` is 1500 then the expression resolves to true, otherwise it returns false (requires that `$mtu` has been initialized).
Does not equal `!=`	`$mtu != 1500` If `$mtu` equals 1500 then the result is false, otherwise it returns true (requires that `$mtu` has been initialized)
Less than <	`$hop-count < 15` Returns true if the left value is less than the right value, otherwise it returns false (requires that `$hop-count` has been initialized).
Less than or equal to <=	`$hop-count <= 14` Returns true if the left value is less than the right value or if the two values are the same, otherwise it returns false (requires that `$hop-count` has been initialized).
Greater than >	`$hop-count > 0` Returns true if the left value is greater than the right value, otherwise it returns false (requires that `$hop-count` has been initialized).
Greater than or equal to >=	`$hop-count >= 1` Returns true if the left value is greater than the right value or if they are the same, otherwise it returns false.
Parenthesis `( )`	`var $result = ( $byte-count * 8 ) + 150;` Used to create complex expressions. Parenthesis function the same way as in a mathematical expression, with the expression within the parenthesis evaluated first. Parenthesis can be nested with the innermost set of parenthesis evaluated first, then the next set, and so on.
And &&	`$byte-count > 500000 && $byte-count < 1000000` The && (and) operator combines two expressions to get one result. If either of the two expressions evaluates to false then the combined expression evaluates to false.

Or `||`	`$mtu-size != 1500 || $mtu-size > 2000` The `||` (or) operator combines two expressions to get one result. If either of the two expressions evaluates to true then the combined expression evaluates to true.
String concatenation _	`var $combined-string = $host-name _ " is located at " _ $location;` The underscore _ is used to concatenate multiple strings together (note that strings cannot be combined using the + operator in SLAX). In the example if `$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;` The `|` operator 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, so the `:=` conversion operator was created. This operator converts a variable from a result tree fragment into a node-set. The script can then tell Junos to search the node-set for the appropriate information and extract it. Only Junos 9.2 and beyond supports this operator (see note next page).

NOTE There is no operator for "not" as there is in other programming languages. Instead, there is a not() function that returns the opposite boolean value of its argument.

NOTE The `:=` operator is only supported in Junos 9.2 and beyond. If you are using an earlier version you can use the node-set() extension function to convert a result tree fragment into a node-set. The node-set() function requires that the "`http://xmlsoft.org/XSLT/namespace`" namespace be declared, and the assigned prefix prepended to the function name. For example, if you assign the namespace to `ext` (`ns ext = "http://xmlsoft.org/XSLT/namespace";`), then you call the function as `ext:node-set(): var $node-set-variable = ext:node-set( $rtf-variable );`

Data Type Conversion

Typically it is not necessary to explicitly convert from one data type into another. The primary exception to this rule is converting from result tree fragment to node-set, but otherwise most conversions occur automatically.

When the script engine comes to an operator or a statement, and the associated date type is not of the correct type, the script engine attempts to automatically convert it. As an example, when the addition operator is encountered, the two arguments are converted into numbers.

The conversion process works in the following way, based on the original data type:

- string: strings which consist entirely of appropriate characters for numeric content are converted into the equivalent number, otherwise they are converted to NaN (not a number). When converting to boolean, empty strings convert to false, and non-empty strings convert to true.
- number: numbers are converted to strings by converting each digit into the appropriate character. The numeric value zero is converted to the boolean value false, all other numeric values are converted to the boolean value true.
- boolean: when converted to strings, booleans become either "true" or "false." The boolean value false is converted to the numeric value of 0. True is converted to the numeric value of 1.
- node-set: a node-set is converted into a string by returning the string value of the first node in the node-set. The string value is the text contents of the node as well as all its child nodes. A node-set converts into a number in a similar fashion. Empty node-sets are converted to the boolean value of false, node-sets with one or more nodes are converted to the boolean value of true.
- result tree fragment: result tree fragments are converted to strings by returning all the text content within the XML data structure.

The following script example shows the automatic conversion process where a string is converted to the needed number data type:

```
/* convert.slax */
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> {

    /* String variable */
    var $numeric-string = "-700";
    /* Number variable */
    var $number = 100;

    /* Output the addition of the two variables to the console */
    <output> $numeric-string + $number;
  }
}
```

When executed, this script displays the number -600 on the console. The script engine automatically converts the string "-700" within the script into the equivalent number -700. The result of the expression of -700 + 100 is then shown.

Try It Yourself: Working with Operators

Create a new script including two variables that are assigned numeric values. Add a third variable and assign it the product of the first two variables. Display the value of the third variable on the console.

Parameters

Parameters are variables whose value is assigned by an external source. Other than their declaration they share the same rules as variables, and you can use them in a similar way.

Default Parameters

Every script begins with six parameters predefined, which are declared within `junos.xsl`. When `junos.xsl` is imported as part of the standard boilerplate, these parameters are imported as well and can be used within the script.

The default parameters provide commonly-used information for scripts:

- `$product`: contains the name of the local Junos device model
- `$user`: is assigned to the name of the user that executed the script
- `$hostname`: stores the local hostname of the Junos device
- `$script`: contains the name of the script that is currently executing
- `$localtime`: stores the local time when the script was executed using the following format: `Tue Jan 20 14:07:33 2009`
- `$localtime-iso`: provides a different format of local time: `2009-01-20 14:07:33 PST`

NOTE In Junos versions prior to 9.6 `$localtime-iso` is named `$localtime_iso`. The old name format will continue to be supported in 9.6 in a deprecated fashion.

Global Parameters

Parameters whose value is set by Junos at script initialization must be defined as global parameters. The default parameters listed above are examples of global parameters. To declare a global parameter, use the `param` statement and provide a name for the parameter. As with variables, parameter names always require a preceding "`$`".

```
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";

/* This is a global parameter */
param $interface;
```

A script can assign a default value to a global parameter. This provides a fallback value in the event that Junos does not give a value to the parameter. If no default value is declared and none is assigned during script processing, then the parameter defaults to an empty string. Here is an example where the `$interface` parameter defaults to "`fxp0`":

```
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";
```

```
/* This is a global parameter with a default value */
param $interface = "fxp0";
```

Command-line Arguments

Global parameters within op scripts are typically used to pass command-line arguments from Junos to the op script. This technique greatly increases the versatility of a script, as scripts can be written to respond differently based on the arguments provided.

Creating Command-line Arguments

Command-line arguments are always expressed as name and value pairs. When a user executes an op script and includes command-line arguments, Junos searches the script for a global parameter of the same name (excluding the dollar sign `$`) and assigns the value from the command-line argument to the matching parameter.

As an example, assume a user entered the following command:

```
user@Junos> op show-interface interface fe-0/0/0.0
```

Based on the command-line entry, Junos searches for a global para- meter named `$interface`. If the parameter is present, then Junos assigns it the string value of "`fe-0/0/0.0`".

Here is an example of a script that utilizes command-line arguments:

```
/* combine-strings.slax */
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";

/* Command-line arguments */
param $string1;
param $string2;

match / {
  <op-script-results> {

    /* Output the command-line arguments to the console */
    <output> $user _ ": Here are your combined strings: " _ $string1 _
$string2;

  }
}
```

This script shows an example of both default parameter use and command-line arguments. `$user` is a default parameter assigned to the name of the user running the script. `$string1` and `$string2` are global parameters which are populated through command-line arguments. Assume the script executed using the following command-line:

```
user@Junos> op combine-strings string1 "Hello" string2 " World!"
user: Here are your combined strings: Hello World!
```

As the example shows, the strings `Hello` and `World!"` were properly assigned to the `$string1` and `$string2` parameters and the `$user` parameter correctly identified the username of `user`.

Op Script Help

Remembering command-line argument names can be burdensome. It is more user-friendly to provide a quick reference as to which command-line arguments are available. Junos provides a way for op scripts to create additions to the Junos CLI help system that remind users which arguments are available for the op script.

By default, Junos users can enter a ? after a command to see the available completions. If a ? is entered following the `op script` command the following is displayed:

```
user@Junos> op combine-strings ?
Possible completions:
 <[Enter]>     Execute this command
 <name>       Argument name
 detail       Display detailed output
 |         Pipe through a command
```

This display does not give any hint of what command-line arguments can be provided to the script. The solution is to use a special purpose global variable named `$arguments` within the op script. Junos automatically looks for this variable when building the help contents for an op script. By following the format of the XML data structure, it is possible to add command-line arguments to the help text.

The structure required for how it uses the `$arguments` variable is:

```
<argument> {
  <name> "ArgumentName";
  <description> "Argument description";

}
```

Take a look at the edited script to see how it is used:

```
/* combine-strings.slax */
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";

/* This variable defines the CLI help text */
var $arguments = {
  <argument> {
    <name> "string1";
    <description> "The first string";
  }
  <argument> {
    <name> "string2";
    <description> "The second string";
  }
}

/* Command-line arguments */
param $string1;
param $string2;

match / {
  <op-script-results> {

    /* Output the command-line arguments to the console */
    <output> $user _ ": Here are your combined strings: " _ $string1 _
$string2;

  }
}
```

Now, notice the difference when a user invokes the Junos CLI help:

```
user@Junos> op combine-strings ?
Possible completions:
 <[Enter]>    Execute this command
 <name>       Argument name
 detail       Display detailed output
 string1      The first string
 string2      The second string
 |            Pipe through a command
```

Try It Yourself: Working with Command-line Arguments

Create a new script with a command-line argument that accepts a number from the user. Include the `$arguments` global variable so that the CLI help output includes the command line argument. Perform a mathematical operation on the command-line argument and output the result to the console. Execute the op script a few times, with a different number provided on the command-line to verify that the result changes.

Conditional If Statements

The script examples to this point have been fairly basic. Conditional code execution allows more flexible functionality with the SLAX `if` statement. Using the `if` statement instructs the script engine to execute segments of code only when certain conditions are met. This means that scripts can react to values instead of only operating on them.

If Statements

The `if` statement consists of two parts: a boolean expression and a conditional code block:

```
if( $mtu == 1500 ) {
  <output> "Jumbo Frames are not enabled";
}
```

In the above example the boolean expression is `$mtu == 1500`. It is expressed within parenthesis immediately following the `if` statement. When the expression evaluates to true then the conditional code block of the `if` statement is executed. When the expression evaluates to false then the script engine skips to the end of the statements code block and continue processing from that point. As an example, if the `$mtu` variable is assigned to the value 1500 then `<output> "Jumbo Frames are not enabled"` is written to the result tree, otherwise this string does not appear in the console output.

Else If and Else Statements

Additional possibilities can be expressed by adding `else if` and/or `else` statements:

```
if( $interface == "fxp0" ) {
  <output> "Out of Band Management";
}
else if( $interface == "lo0" ) {
  <output> "Loopback Interface";
}
else {
  <output> "Other";
}
```

In this case, the script engine checks each boolean expression sequentially. The first expression that evaluates to true has its code block executed. If neither the `if` nor the `else if` evaluate to true, then the `else` code block is executed (when present). In all cases, the script engine executes a maximum of one conditional code block. If multiple boolean expressions evaluate to true, the script engine only applies the first.

Conditional Operation Example

Here is a script that shows conditional operation. It outputs the value of one default parameter, with the desired parameter being chosen through a command-line argument:

```
/* output-parameter.slax */
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";

/* This shows the parameter argument in the CLI help output */
var $arguments = {
  <argument> {
    <name> "parameter";
    <description> "Enter name of parameter, e.g. $user";
  }
}

/* Command-line argument */
param $parameter;

match / {
  <op-script-results> {
    /* Output the selected parameter to the console */
    if( $parameter == "$user" ) {
      <output> "$user = " _ $user;
    }
    else if( $parameter == "$hostname" ) {
      <output> "$hostname = " _ $hostname;
    }
    else if( $parameter == "$product" ) {
      <output> "$product = " _ $product;
    }
    else if( $parameter == "$script" ) {
      <output> "$script = " _ $script;
    }
    else if( $parameter == "$localtime" ) {
      <output> "$localtime = " _ $localtime;
    }
    else if( $parameter == "$localtime-iso" ) {
      <output> "$localtime-iso = " _ $localtime-iso;
    }
    /* If nothing matches then give this message instead */
    else {
      <output> "This is not a valid default parameter: " _ $parameter;
    }
  }
}
Now let's see this script in action:
user@Junos> op output-parameter parameter $user
```

```
$user = user

user@Junos> op output-parameter parameter $script
$script = output-parameter.slax
```

Conditional Variable Assignment

One common use for the `if` statement is to conditionally assign variable values. Because a variable's initially assigned value cannot be reassigned, it is prudent to be very selective in the value bound to a variable. This can be done by declaring that a variable must have its value assigned by an `if` statement:

```
var $user-type = {
  if( $user == "john" ) {
    expr "operator";
  }
  else if( $user == "tom" ) {
    expr "admin";
  }
  else {
    expr "unknown";
  }
}
```

Observe the following points in the above code. First, in order to conditionally assign a variable the entire `if` statement and any attached `else if` and `else` statements must all be enclosed within curly braces { }. Second, note the use of a new statement `expr`, which is used to write text to the result tree. What the code above actually does is write a conditionally selected string to the result tree. But this string is redirected to the `$user-type` variable making it a result tree fragment.

Suppose the `$user` default parameter equals "`john`" then the "`operator`" string is written to the variable `$user-type` as a result tree fragment. Having a data type of result tree fragment in place of a string does not cause any problems because the script engine automatically converts the data type result tree fragment to a string whenever necessary. Because of this, the script can treat the result tree fragment as if it was just a normal string variable.

Try It Yourself: Conditionally Assigning Variable Values

Create a new script with a command-line argument that can be set to either + or - , signifying the mathematical operation to perform. Create a variable that is assigned conditionally, based on the value of the command-line argument. If the command-line argument is specified as a + then two values should be added together and assigned to the variable. If the command-line argument is specified as a - then subtraction should be performed between the two values and assigned to the variable. Output the result to the console.

Named Templates

The examples provided so far have included only the main template. This is appropriate given their simple nature. However, as the complexity of a script increases it becomes more advantageous to modularize it by removing some of the functionality from the main template and placing the code into named templates instead.

A named template is a segment of code which can be called from anywhere in the script by referring to its name. When this occurs, the script engine shifts its focus to the code within the selected template until the script completes. Then the script

engine returns to the calling template and continues processing from where it left the script.

Named templates can greatly enhance scripts in the following ways:

- code re-use: if a particular code stanza has to be repeated multiple times throughout your script then it makes sense to place it within a named template instead. This reduces the size of your script and simplifies changes.
- self-documentation: named templates with descriptive names clarify the script actions. A script that is written in this way is simpler to read and understand than one in which all the operations are performed in a single large main template.
- recursion: a useful capability of named templates is their ability to call themselves. By looping through a section of code as many times as necessary, you can program the script to reach a specific end goal.

Named Templates Syntax

To create a named template you use the `template` statement, give it a name, and provide its curly brace enclosed code block:

```
template example-template {
  /* Template code goes here */
}
```

To call a template use the `call` statement and include the name of the desired template:

```
call example-template;
```

Here is an example of a script that uses a named template:

```
/* show-user.slax */
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> {

    /* Call the display-user template */
    call display-user;

  }
}

/* This template outputs the username to the console */
template display-user {
  <output> "Your user name is " _ $user;
}
```

In this example, the main template calls the `display-user` named template. The called template is then executed, causing the `<output>` element to be written to the result tree along with its included string. Although this is a simple example, it highlights a significant fact about named templates: any elements that are written to the result tree by a named template are inserted at the point the template is called.

The main template code writes the <op-script-results> element to the result tree, but before doing so it calls the display-user template. The display-user template then provides the instructions to include the <output> element as a child element of <op-script-results>. The final result tree sent to Junos is:

```
<op-script-results> {
  <output> "Your user name is " _ $user;
}
```

Template Parameters

Template parameters are similar to global parameters – their value is set outside of the template. However, rather than being set by Junos they are set by the script code when it codes the named template.

To declare template parameters, include them within parenthesis following the template name:

```
template display-user( $full-name ) {
  /* Template code goes here */
}
```

A default value can be provided in the same way as a global parameter. If the script code provides no default value, and the template code does not assign a parameter value, when the template is called, then the parameter is set to an empty string.

```
template display-user( $full-name = "John Doe" ) {
  /* Template code goes here */
}
```

NOTE An alternate, more verbose, method to declare parameters is to use the param statement in the lines immediately following the template name.

Parameter values are assigned within the same statement that calls the named template. The assignments are made by name, not by position:

```
call display-user( $full-name = "Jane Doe" );
```

This instructs the script engine to call the display-user template and to give its $full-name parameter a value of "Jane Doe".

NOTE An alternate, more verbose, method to assign template parameter values when calling a named template is to use the with statement in the following manner:

```
call display-user {
  with $full-name = "Jane Doe";
}
```

This script example shows how to use template parameters:

```
/* show-time.slax */
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> {

    /* Call the display-time template and set the $format parameter */
    call display-time( $format = "normal");
```

```
  }
}

/* Output the localtime to the console in either iso (default) or normal
format */
template display-time( $format = "iso" ) {
  if( $format == "iso" ) {
    <output> "The iso time is " _ $localtime_iso;
  }
  else {
    <output> "The time is " _ $localtime;
  }
}
```

In this example, the `display-time` template shows the execution time of the script in either the default ISO format or the normal format. When the script is called the `$format` parameter is set to "normal" resulting in the normal time being displayed on the console.

This op script would be more useful if it allowed the user to choose the format. The following adds a command-line argument for `$format` that lets the user do so:

```
/* show-time.slax */
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";

/* This is imported into the Junos CLI help text */
var $arguments = {
  <argument> {
    <name> "desired-format";
    <description> "Choose either iso or normal";
  }
}

/* Command-line argument */
param $desired-format;

match / {
  <op-script-results> {

    /* Call the display-time template and set the $format parameter */
    call display-time( $format = $desired-format );
  }
}

/* Output the localtime to the console in either iso (default) or normal
format */
template display-time( $format = "iso" ) {
  if( $format == "iso" ) {
    <output> "The iso time is " _ $localtime_iso;
  }
  else {
    <output> "The time is " _ $localtime;
  }
}
```

This modified script includes a new `desired-format` command-line argument, allowing the user to choose in which format to display the time. When the `display-time` template is called the `$format` parameter is set to the value of the `$desired-format` global parameter.

Here is an example of the output:

```
user@Junos> op show-time display-format iso
The iso time is 2009-05-12 21:01:10 PDT

user@Junos> op show-time display-format normal
The time is Tue May 12 21:01:13 2009
```

NOTE If the calling template includes variables or parameters with the same name as the parameter of the called template, then the parameter can be listed without applying an assignment. The script engine automatically sets it to the value of the variable or parameter within the calling template. For example:

```
var $display-string = "Example String";
call show-string( $display-string );
```

SHORT CUT There is an easier way to write the script on the previous page. Remember that global parameters and variables are accessible in the entire script, not just in the main template. This means that it is not necessary to pass the `$format` value to the `display-time` template as a template parameter. Instead, the `display-time` template can simply use the global parameter. The Appendix included in the PDF version of this book provides an example that uses the global parameter. The script operates in the same manner as the preceding one, but in this case the named template accesses the global parameter instead of relying on the main template to pass the format as a template parameter.

Redirecting the Result Tree

Creating templates that perform a subset of the script code is useful for code modularization and self-documentation. Additionally, there are many times when the calling template can process a desired result from the called named template. In this scenario, the called template is designed to perform a specific operation and then return the result. Named templates (unlike functions in other languages) do not have a direct mechanism for returning values; however, they are able to write to the result tree, and it is possible to redirect the result tree to a variable. Doing this allows called templates to effectively return a string value to the calling template by writing to the result tree. The calling template then redirects the result tree output into a variable. Here is an example of how to do this:

```
/* show-day.slax */
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> {

    /* get-day-of-week returns the day - assign it to $day-of-week */
    var $day-of-week = { call get-day-of-week(); }

    /* Output day string to the console */
    <output> $day-of-week;
  }
}

/* Extract the day of week string from the $localtime global parameter */
template get-day-of-week {
  /* Write the first three characters of the $localtime to the result tree */
  expr substring( $localtime, 1, 3 );
}
```

The `get-day-of-week` template extracts the day string from `$localtime` and writes it to the result tree. The calling template can then redirect the result tree output and store it within a variable instead. As seen in the script, the syntax used to accomplish this is similar to the syntax used for conditional variable assignment. In both cases the script requires the use of curly braces around the code, which writes to the result tree:

```
var $day-of-week = { call get-day-of-week(); }
```

By extracting the day string from `$localtime` and writing it to the result tree through a named template, the script can call this named template multiple times. The named template can also be easily copied from one script into another that requires similar functionality. To retrieve the day string the script takes advantage of the `substring()` function within the `get-day-of-week` template. The last section of this chapter covers using functions.

Try It Yourself: Working with Named Templates

Create a new script that contains a named template. The template should write a string to the result tree. Redirect this into a variable in the calling template and output the variable value to the console.

Functions

Functions are coded procedures within the script engine. A script can invoke functions, which take arguments, perform a specific action, and return the result. This process might sound similar to a named template, but there are large differences between the two:

- A named template is actual script code, whereas a function is part of the underlying Junos operating system itself.
- Values are provided to named templates through the use of parameters, which are assigned by name, but functions use arguments where a precise order is mandated.
- Functions actually return results, whereas named templates can only write to the result tree and have that result tree fragment redirected to a variable in the calling function.

The syntax of functions differs from that of named templates as well. The `call` statement is not used; only the function name is provided and the required arguments specified within parenthesis:

```
expr substring( $localtime, 1, 3 );
```

Functions return values, as an example the substring() function returns a string. The above code writes the string value to the result tree. The script code could assign the string value to a variable instead using the following syntax:

```
var $day-string = substring( $localtime, 1, 3 );
```

Note the difference between assigning a value to a variable from a named template versus from a function:

```
var $day-of-week = { call get-day-of-week(); }
```

String Functions

String functions are used regularly within scripts. Table 3.2 lists some of the most common and useful of the string functions.

Table 3.2 **String Functions**

Function	Example Explanation
`substring( string-value, starting-index, length )`	`var $hello-string = substring( "Hello World", 1, 5 );` Takes a starting string and returns the substring that begins at the specified index and extends for the given length. This example results in the `$hello-string` being assigned the string value "`Hello`". In SLAX, indexes always begin with 1.
`substring-before( string-value-1, string-value-2 )`	`var $hello-string = substring-before( "Hello World", " " );` Returns a substring of `string-value-1`, but in this case the size of the substring is determined by the location of `string-value-2` within `string-value-1`. The function returns the entire portion of `string-value-1` up to `string-value-2`. The example results in `$hello-string` being assigned the string value "`Hello`".
`substring-after( string-value-1, string-value-2 )`	`var $world-string = substring-after( "Hello World", " " );` Returns the portion of `string-value-1` which comes after `string-value-2` [i.e.,the reverse logic of `substring-before()`]. This example sets `$world-string` to the string value "`World`".

`contains( string-value-1, string-value-2 )`	`if( contains( $interface-name, "ge-" ) ) {`
	`  <output> "The interface is Gigabit-Ethernet";`
	`}`
	
	Returns a boolean value of true or false. If `string-value-1` contains `string-value-2` then it returns true, otherwise it returns false. The example shows the `contains()` function. This code uses it to determine if an interface is a ge or not based on the presence of the second string "`ge-`" in the string `$interface-name`. If the returned value is true then a string is written to the result tree.

`starts-with( string-value-1, string-value-2 )`	`if( starts-with( $interface-name, "ge-" ) ) {`
	`  <output> "The interface is Gigabit-Ethernet";`
	`}`
	
	Returns a boolean value of true if `string-value-1` begins with `string-value-2`, otherwise it returns false. In the example, if `$interface-name` begins with "`ge-`" then a string is written to the result tree.

`string-length( string-value )`	`expr string-length("ospf");` Returns the number of characters within the string. The example causes the value 4 to be written to the result tree.
`translate( string-value, from-string, to-string )`	`var $new-string = translate( $string, "abcdefghijklmnopqrstuvwxyz", "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );` The `translate()` function translates the characters within the `string-value`, where the function translates any matching characters within `from-string` to their corresponding characters in `to-string` and returns the result. This example translates a string into upper-case.

MORE? This chapter only covers a portion of the available functions. Additional functions are in the *Configuration and Diagnostic Automation Guide* at www.juniper.net/techpubs.

jcs:printf()

Printing unformatted output to the screen is sufficient for some scripts, but in many cases it is more user-friendly to have formatted script output where each line follows the same column spacing. The `jcs:printf()` function is used in op scripts for this purpose. It returns a string based on the formatting instructions and values provided in the arguments.

NOTE The `jcs:printf()` function does not output directly to the console, it only returns a formatted string. This string can then be output to the console as desired.

The syntax for `jcs:printf()` is the following:

```
jcs:printf( "expression", value1, value2, ..., valuex );
```

The string expression contains embedded placeholders that indicate where each value should be inserted, as well as the format in which they should be placed. Here is an example:

```
<output> "12345678901234567890123456789";
<output> jcs:printf("%-10s%-10s","OSPF","ISIS" );
<output> jcs:printf("%10s%10s", "OSPF", "ISIS" );
```

There are two embedded placeholders in the expression of each of these `jcs:printf()` function calls. An embedded placeholder is indicated by a `%` followed by any flags, the width of the field, and the value type. In the first case:

```
<output> jcs:printf("%-10s%-10s", "OSPF", "ISIS" );
```

The `%-10s` indicates that a string value is inserted in a 10 space column that is left justified. This is repeated twice, once for "`OSPF`" and once for "`ISIS`". In the second case:

```
<output> jcs:printf("%10s%10s", "OSPF", "ISIS" );
```

The size and number of fields are the same but they lack the – flag, which causes them to be right justified. Here is the output that is displayed based on the three lines of code above:

```
user@Junos> op show-pretty-output
12345678901234567890123456789
OSPF       ISIS
      OSPF       ISIS
```

Through proper usage of the `jcs:printf()` function an op script can produce output that looks just as well formatted as the normal Junos operational commands.

Try It Yourself: Working with Functions

Create a new script with a variable assigned to the value "Juniper Networks". Output the following to the console on separate lines:

1. The variable value
2. The variable value - right justified in a 20 space field
3. The string length of the variable
4. The substring before the space
5. The string converted entirely into uppercase

Chapter 4

Communicating with Junos

Chapters 2 and 3 covered most of the necessary SLAX syntax and statements used to build functional scripts, but the real potential of Junos automation is accomplished through direct communication with Junos devices. This chapter discusses the process that allows the script to interact with a Junos device, including retrieval of operational information, writing to the syslog, and working with the configuration.

Invoking Operational Commands

Most operational commands that can be executed manually on the Junos CLI prompt can also be invoked within an automation script. Junos processes the commands in the same manner as if they were run from the CLI. However, Junos generates their output in XML and provides it to the script. The script can then parse the results and perform any desired actions based on the gathered information.

Junos XML API

Invoking operational commands in Junos is possible through use of the Junos XML API. The script code sends API Elements to Junos, which then processes the received elements, performs the associated actions, and returns the results to the script.

Many operational commands are mapped directly to an API Element. For example, the "`clear interfaces statistics`" CLI command is mapped to `<clear-interfaces-statistics>`. To see the list of mapping between CLI commands and API Elements consult the Junos XML Operational API Reference Guide (some examples are in Table 4.1). Operational commands that do not have a specific API Element can be executed by using the `<command>` element with the CLI command as its text content:

```
<command> "show route";
```

Table 4.1 API Element Examples

API Element	CLI Command
`<get-configuration>`	`show configuration`
`<get-isis-adjacency-information>`	`show isis adjacency`
`<get-ospf-interface-information>`	`show ospf neighbor`
`<get-bgp-neighbor-information>`	`show bgp neighbor`
`<get-chassis-inventory>`	`show chassis hardware`
`<get-interface-information>`	`show interfaces`
`<get-route-information>`	`show route`
`<get-software-information>`	`show version`

jcs:invoke()

The Script code sends API elements to Junos by calling the `jcs:invoke()` function and providing the element as an argument. The API Element can be expressed either through a result tree fragment variable, or it can be a string containing the API Element's name. Any results from `jcs:invoke()` are returned as a node-set and should be assigned to a variable:

```
var $clear-statistics-rpc = <clear-interfaces-statistics-all>;
var $results = jcs:invoke( $clear-statistics-rpc );

or
var $results = jcs:invoke( "clear-interfaces-statistics-all" );
```

In both cases, the `clear interfaces statistics all` command is invoked, and any XML results are assigned to the `$results` variable. The difference between specifying the API Element in XML versus providing the string name is that XML attributes, text content, and child elements can only be included when the API Element is expressed as an XML data structure assigned to a variable that is passed to `jcs:invoke()`. Shown here where the interface to be cleared is provided as a command-line argument:

```
/* clear-statistics.slax */
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";

/* This is imported into the Junos CLI help text */
var $arguments =  {
          <argument> {
            <name> "interface";
            <description> "Clear the specified interface statistics";
          }
        }

/* Command-line argument */
param $interface;

match / {
  <op-script-results> {

    /* Junos XML API Element to clear specific interface statistics */
    var $clear-statistics-rpc = <clear-interfaces-statistics> {
                <interface-name> $interface;
              }

    /* Send XML API Element to Junos through jcs:invoke() */
    var $results = jcs:invoke( $clear-statistics-rpc );

    /* Copy XML contents of $results to the result tree */
    copy-of $results;

  }
}
```

In this example, the `<clear-interfaces-statistics>` API Element is used to clear the statistics of the interface that is specified within the command-line of the op script. The API Element is sent to Junos by the `jcs:invoke()` function and the results of the operation are stored in the `$results` variable.

Finally, a new statement can be seen in the script: `copy-of`. The `copy-of` statement is used to copy the contents of a result tree fragment variable or a node-set to the result tree. The script code above does this to communicate any error messages. If `<clear-interfaces-statistics>` executes successfully then no result is provided, but if there is an error then a `<xnm:error>` element is returned with a `<message>` child element. `<xnm:error>` is a valid op script result tree element, so copying it to the result tree causes the included `<message>` to be displayed to the console as an error message:

```
user@Junos> op clear-statistics interface ge-13/0/0
error: device ge-13/0/0 not found
```

NOTE The Junos device processes the received API Elements from `jcs:invoke()` immediately rather than waiting for the script to terminate as with the result tree.

ALERT! All Junos commands and configuration requests performed through op scripts follow the same permission process as if the command were entered at the CLI. In both cases, Junos checks permission levels to verify that the user has access to that command or configuration hierarchy.

Try It Yourself: Invoking Junos Operational Commands

Following the example of the clear-statistics op script shown in this section, create an op script that reboots the system. (Hint: The XML API Element needed is `<request-reboot>`).

Retrieving Data

In the last section, the clear statistics script used the `copy-of` statement to send the XML contents of the `$results` variable to the result tree. But what exactly were those contents? The simplest way to see what XML results are returned from `jcs:invoke()` is to run the command manually with `| display xml` appended:

```
user@Junos> op clear-statistics interface ge-13/0/0 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.0R4/junos">
  <xnm:error xmlns="http://xml.juniper.net/xnm/1.1/xnm">
    <source-daemon xmlns="">
      ifinfo
    </source-daemon>
    <message xmlns="">
      device ge-13/0/0 not found
    </message>
  </xnm:error>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

When `jcs:invoke()` returns its XML result, it assigns the child element of `<rpc-reply>` to the `$results` variable. In this case that is `<xnm:error>`. In the clear statistics script the returned XML data structure is just copied into the result tree. But consider the following modification, which allows a success message to be displayed when no error occurs:

```
/* clear-statistics.slax */
                        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";

                        /* This is imported into the Junos CLI help text */
                        var $arguments =  {
                                  <argument> {
                                    <name> "interface";
                                    <description> "Clear the specified interface statistics";
                                  }
```

```
        }

/* Command-line argument */
param $interface;

match / {
  <op-script-results> {

    /* Junos XML API Element to clear specific interface statistics */
    var $clear-statistics-rpc = <clear-interfaces-statistics> {
                <interface-name> $interface;
              }

    /* Send XML API Element to Junos through jcs:invoke() */
    var $results = jcs:invoke( $clear-statistics-rpc );

    /*
     * Check if <xnm:error> is part of $results, if it is then
     * copy the output to the console. Otherwise show the success
     * message.
     */
    if( $results/..//xnm:error ) {
      copy-of $results;
    }
    else {
      <output> "Statistics Cleared";
    }
  }
}
```

The change here is the addition of an `if` statement with the following test: `$results/..//xnm:error`. This is an example of a location path that pulls data from the XML data structure and assigns it to the `$results` variable. If a `<xnm:error>` element is present in the results then the expression evaluates to true and the XML contents of `$results` are copied to the result tree. Otherwise `"Statistics Cleared"` is output to the console showing that the op script has successfully cleared the interface statistics.

Location Paths

Location paths are used to extract data from an XML structure. They follow a fixed syntax that dictates which nodes should be retrieved and included within the output. The results of a location path are communicated as a node-set that can be assigned to a variable or used within a SLAX statement such as `if` or `for-each`.

This example helps better illustrate the purpose and usage of location paths. Consider the following XML data structure:

```
<system-uptime-information xmlns="http://xml.juniper.net/junos/9.0R4/
junos">
    <current-time>
      <date-time junos:seconds="1242673659">2009-05-18 12:07:39 PDT</
date-time>
    </current-time>
    <system-booted-time>
      <date-time junos:seconds="1242424838">2009-05-15 15:00:38 PDT</
date-time>
      <time-length junos:seconds="248821">2d 21:07</time-length>
    </system-booted-time>
    <protocols-started-time>
```

```
     <date-time junos:seconds="1242424912">2009-05-15 15:01:52 PDT</date-
time>
     <time-length junos:seconds="248747">2d 21:05</time-length>
   </protocols-started-time>
   <last-configured-time>
     <date-time junos:seconds="1242424900">2009-05-15 15:01:40 PDT</date-
time>
     <time-length junos:seconds="248759">2d 21:05</time-length>
     <user>root</user>
   </last-configured-time>
   <uptime-information>
     <date-time junos:seconds="1242673659">12:07PM</date-time>
     <up-time junos:seconds="248851">2 days, 21:07</up-time>
     <active-user-count junos:format="2 users">2</active-user-count>
     <load-average-1>0.00</load-average-1>
     <load-average-5>0.00</load-average-5>
     <load-average-15>0.00</load-average-15>
   </uptime-information>
 </system-uptime-information>
```

This is the XML output of the operational command `show system uptime` (the API Element is `<get-system-uptime-information>`). When this command is executed by `jcs:invoke()` the `<system-uptime-information>` element is returned as the XML result. Once this XML data has been retrieved and placed into a node-set variable it is possible for location paths to extract the embedded information.

Using location paths is similar to locating files within a file system. The search starts at a particular reference point and the use of the forward slash `/` indicates that the search path will move to a child of the current reference point. As an example, assume that `$result` is the name of the node-set variable which has been assigned the above XML data structure. If the boot-time value is desired then it can be retrieved with the following location-path:

```
$results/system-booted-time/date-time
```

This element node could be assigned to another variable:

```
var $date-time = $results/system-booted-time/date-time;
```

Or the text value of the date-time element node could be output to the console:

```
<output> $results/system-booted-time/date-time;
```

Reviewing the location path shown above, `$results` has a default reference point of the `<system-uptime-information>` element node. This is the starting context node from which all location paths defined using this variable are based.

TIP An easy way to verify the starting reference point of a variable is to use the `name()` function. For example: `<output> name( $results );` would display the XML element name to the console.

The / indicates a location path step. When taking a step, the default action is to look at the child of the context node. The location path example specifies `<system-booted-time>` so all the child nodes of `<system-uptime-information>` that are named `<system-booted-time>` are selected (in this case, only one). Next the second / indicates another location path step. The context node within the location path changes at each step, now it has become the `<system-booted-time>` element. Once again the default step is used to search for a child element named `<date-time>` (there is only one in this output). This is the end of the location path so the `<date-time>` element

node is returned. Examples of location paths using the prior XML data structure:

Was the router booted this year?

```
If( contains( $results/system-booted-time/date-time, "2009" ) ) {
  /* conditional code goes here... */
}
```

How many users are currently online?

```
var $user-count = $results/uptime-information/active-user-count;
```

NOTE A // can be used instead of a / if the location path should match on the desired child node, no matter how deep into the hierarchy it appears. For example `$results//active-user-count` would return the same output as `$results/uptime-information/active-user-count`.

Parent Axis

By default, the child axis is used for each location path step, but what if the script needs to go in the opposite direction?

Assume the following has been set in the starting-template:

```
var $results = jcs:invoke( "get-system-uptime-information" );
var $date-time = $results/system-booted-time/date-time;
call display-boot-time( $date-time );
```

The `display-boot-time` template has been called with its `$date-time` parameter set to the `$date-time` variable in the calling template, which consists of a single element node `<date-time>`. But the `display-boot-time` template is intended to display both the `<date-time>` text value as well as its sibling element `<time-length>`. In order to do this, there must be some way to retrieve `<time-length>` while starting with `<date-time>` as a reference point.

The solution is to use the parent axis instead of the default child axis. Here is an example of how the template could be written to output both of the element node values:

```
template display-boot-time( $date-time ) {
  <output> "Here is the boot time: " _ $date-time;
  <output> "Here is the time length: " _ $date-time/../time-length;
}
```

This template outputs both the <date-time> element node value to the console and the <time-length> element node value. The `$date-time` parameter is initialized to the `<date-time>` node so the first output line can be printed by referring to the variable itself. For the second line a new location path operator is introduced: the `..` parent axis abbreviation. Just like with a file system `cd ..` command the `..` causes the location path to search the parent of the context node rather than its children. So the path goes from `<date-time>` to the `<system-booted-time>` parent. A following `/` indicates another step, this time using the default child axis, after which the element node `<time-length>` is selected.

Attribute Axis

The child and parent axis are the most commonly used axes within location paths. The next most needed axis is the attribute axis. It is through this axis that attribute nodes can be extracted from XML data structures.

Take a look at the <system-uptime-information> XML output again and note the number of junos:seconds attributes that are included. For every time output there is also a corresponding junos:seconds attribute applied to the element. This attribute contains the second value as an alternative to the complete date/time string.

If this second value is desired instead of the more verbose string then it can be retrieved through the attribute axis, which is specified by using the @ abbreviation:

```
var $results = jcs:invoke( "get-system-uptime-information" );
var $date-time = $results/system-booted-time/date-time/@junos:seconds;
<output> "Boot time seconds: " _ $date-time;
```

In this case an additional location path step has been added. Rather than stopping at the date-time node, the location path goes further and retrieves the junos:seconds attribute node by using the attribute axis as denoted by the @. As a result, the $date-time variable is assigned to the milliseconds value, which is then output to the console.

MORE? There are thirteen different axes that can appear in a location path but most are rarely used. To read about the other location path axes consult the XPATH specification: http://www.w3.org/TR/xpath#axes/.

Predicates

It is often necessary to be more selective about which nodes are extracted than is shown in the prior examples. If there are multiple occurrences of a node of the same name then location paths, based on the syntax above, would return all of them instead of only a single node. If the goal is to process all of the returned nodes in the same manner then that might be the desired behavior. But if the intent is to only extract a specific node among identically named nodes then a predicate must be used to indicate this within the location path.

For example, the output of show interface terse displays all the interfaces available on the Junos device. The corresponding <get-interface-information> API Element returns the same list of interfaces. That might be what a script needs, or it might not. It's important to be aware of the default behavior – which is to return all of the nodes with the same name – and to know how to sharpen the location path using predicates when a more specific query is desired.

Predicates are filters that prevent non-matching nodes from being included in the location path result. The predicate expression is enclosed within brackets [ ] for example:

```
var $get-interface-rpc = <get-interface-information> {
            <terse>;
          }
var $results = jcs:invoke( $get-interface-rpc );
var $ge-node = $results/physical-interface[name=="ge-1/0/0"];
```

Here is the XML output of <get-interface-information> <terse>:

```
<interface-information>
   <physical-interface>
     <name>ge-1/0/0</name>
     <admin-status>up</admin-status>
     <oper-status>up</oper-status>
     <logical-interface>
       <name>ge-1/0/0.0</name>
       <admin-status>up</admin-status>
```

```
      <oper-status>up</oper-status>
      <address-family>
        <address-family-name>inet</address-family-name>
        <interface-address>
          <ifa-local junos:emit="emit">1.1.1.1/24</ifa-local>
        </interface-address>
      </address-family>
      <address-family>
        <address-family-name>iso</address-family-name>
      </address-family>
      <address-family>
        <address-family-name junos:emit="emit">mpls</address-family-name>
      </address-family>
    </logical-interface>
  </physical-interface>
  ... repeated for every interface
</interface-information>
```

If a script does not specify a predicate, the location path `$results/physical-interface` returns a node-set with the `<physical-interface>` element nodes for ALL the interfaces in the Junos device. But because the script includes the `[name=="ge-1/0/0"]` predicate, the location path is instructed to only include the `<physical-interface>` element nodes in the returned node-set if they have a child named `<name>` whose value is "`ge-1/0/0`". As a result, the location path only returns a single <physical-interface> element node: the `ge-1/0/0` interface.Multiple predicates can be included within a location path. When more than one predicate is present they are read from left to right. The expressions within all of the predicates must evaluate to true or the currently compared node is not included in the location path result.

This op script shows how predicates can be used in location paths to sharpen the results. Processing the script displays the admin status of only a single interface on the console:

```
/* show-admin-status.slax */
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";

/* This is imported into the Junos CLI help text */
var $arguments =  {
          <argument> {
            <name> "interface";
            <description> "Show admin status of interface";
          }
        }

/* Command-line argument */
param $interface;

match / {
  <op-script-results> {

    /* Junos XML API Element for show interface terse */
    var $get-interface-rpc = <get-interface-information> {
                <terse>;
              }

    /* Retrieve the results of the API request */
```

```
    var $results = jcs:invoke( $get-interface-rpc );

    /* Assign all matching nodes from the results to the $admin-status
variable */
    var $admin-status = $results/physical-interface[name==$interface]/
admin-status;

    /* Output the interface admin status to the console */
    <output> "The admin status of " _ $interface _ " is " _ $admin-status;
  }
}
```

The op script allows the interface to be chosen through a command-line argument. Then, after retrieving the `<get-interface-information>` output from the `jcs:invoke()` function, the admin status of only the selected interface is extracted through a location path with a predicate included. The output of the script shows:

```
user@Junos> op show-admin-status interface ge-1/0/0
The admin status of ge-1/0/0 is up
```

Location Path Operators, Abbreviations, and Wildcards

Table 4.2 lists the operators used within location paths.

Table 4.2 Location Path Operators

Name Code	Example Explanation
Location Path Step `/`	`var $host-name = $results/system/host-name;` Each `/` represents one step in the XML hierarchy. The direction of the step depends on the axis in use, with the default being the child axis.
Multiple Steps `//`	`var $host-name = $results//host-name;` The `//` skips over multiple steps in the hierarchy. This example is the same as the previous example but the `<system>` element node is skipped since the `//` operator will search through zero or more steps. If there were `<host-name>` element nodes in other hierarchy levels they would be returned by this location path as well.
Parent Axis `..`	`var $errors = $results/..//xnm:error;` The `..` is an abbreviation for the parent axis. It indicates that the parent axis should be searched instead of the default child axis.
Attribute Axis `@`	`var $changed = $configuration//@changed;` The `@` sign is an abbreviation for the attribute axis, indicating that Junos should search the attribute axis instead of the default child axis.
Wildcard Match `*`	`var $user-children = $configuration/system/login/user/*;` The wildcard matches all nodes along the given axis (by default the child axis). In this example it would match all of the child nodes for all `<user>` element nodes.

Predicates [ ]	`var $ge-interface = $configuration/interfaces/interface[ starts-with(name, "ge" ) ];` Predicates are bounded by [ ]. If their expression evaluates to false then the node is not included in the location path result.
Context Node .	`var $ge-interface = $configuration/interfaces/interface/name[ starts-with(., "ge") ];` A period can be used within a predicate to indicate that the expression should use the context node value. In this example the `<name>` element node itself is evaluated by the `starts-with()` function since the `.` is used as an argument.

Try It Yourself: Retrieving Information from Junos

Create a script similar to the `show-admin-status.slax` example script above, but instead of the Admin Status report the MTU of a physical interface to the screen. The interface to be displayed should be selected through a command-line argument.

Looping with For-each

It can be necessary to loop through several returned nodes within a node-set as a result of location paths. The `for-each` statement does this. It instructs the script engine to execute a code block for the first node and then loop back through the code block for every node until the node list is exhausted, at which point the script engine continues past the `for-each` statement.

The syntax of a `for-each` loop is similar to an `if` statement but the contents within its parenthesis consist of a location path or node-set variable rather than a conditional expression:

```
for-each( $results/physical-interface/mtu ) {
  /* looped code */
}
```

The `$results/physical-interface/mtu` location path is evaluated and a list of nodes returned. The `for-each` code block is then executed for each node in the node list. Here is a script that displays the mtu of all physical interfaces in the router:

```
/* show-mtu.slax */
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> {

    /* Send Junos XML API Element via jcs:invoke */
    var $results = jcs:invoke( "get-interface-information" );

    /* Create node list based on location path, loop through each node */
    for-each( $results/physical-interface/mtu ) {

      /* Output the MTU for all interfaces that don't have Unlimited MTU */
      if( . != "Unlimited" ) {
        <output> "The MTU for " _ ../name _ " is " _ .;
      }
```

```
    }
  }
}
```

After the <get-interface-information> API Element is used to retrieve the desired information, a location path based on the returned XML data is provided to the for-each statement. This location path provides a result of all the <mtu> element nodes that are children of a <physical-interface> element node. In other words, all the <mtu> nodes of physical interfaces are returned.

This node list is submitted to the for-each loop and each time through the loop the mtu for the interface is output to the screen. An if statement specifies that interfaces without a typical MTU (their MTU is listed as "Unlimited") are not shown.

NOTE This same behavior is enforced through a location path predicate instead: `$results/physical-interface[ mtu != "Unlimited" ]`

The code example uses the . abbreviation to refer to the context node. Each time through the loop the context node changes as a new node is selected from the node list. The conditional expression `. != "Unlimited"` is testing if the current <mtu> element node has a value of "Unlimited" or not. When it does, then its MTU is not shown on the console.

With the addition of the for-each statement a new reference point, besides variables, is now available for location paths. Each iteration through the loop works with a different context node. This context node is the reference point for any location paths that are not based on a variable. Look again at this line from the script:

```
<output> "The MTU for " _ ../name _ " is " _ .;
```

The . at the end of the sentence refers to the current <mtu> element node. Concatenating the element node to a string causes its value to be included. Also, note how the ../name location path is included without any attached variable. Instead, the ../name location path is compared against the <mtu> context node. This causes the path to search first the parent <physical-interface> node of the <mtu> node and then to select its <name> child node. The result is that the text name of the physical interface is included in the output string.

Try It Yourself: Retrieving Information from Junos

Create a script that displays the logical interface MTU of all interfaces within your Junos device.

Interactive Input

Chapter 3 demonstrated how input can be given to the op script through command-line arguments. Beginning in Junos 9.4 it is also possible to accept this input interactively within the script itself by using the jcs:get-input() function. In Junos 9.6 the jcs:get-secret() function was also added to prompt for input while hiding the entered answer from the user.

jcs:get-input() / jcs:input()

The jcs:get-input() function (known as jcs:input() in Junos 9.4 and 9.5) causes a

prompt to be displayed on the console while the script engine pauses for the user to type a response that is terminated by the enter key. A prompt string is specified as the only argument to the function, the entered answer is returned as a string:

```
var $user-input = jcs:get-input("Enter your favorite protocol: ");
```

NOTE `jcs:input()` requires Junos 9.4 or above. Starting in Junos 9.6, use `jcs:get-input()` instead.

A good use for `jcs:get-input()` is to gather needed information from missing command-line arguments. As an example, here is a modified script that displays the admin status of an interface. It has a single command-line argument of `interface`. If the command-line argument is missing then the script uses `jcs:get-input()` to learn the interface value:

```
/* show-admin-status.slax */
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";

/* This is imported into the Junos CLI help text */
var $arguments =  {
          <argument> {
            <name> "interface";
            <description> "Show admin status of interface";
          }
        }

/* Command-line argument */
param $interface;

match / {
  <op-script-results> {

    /* Junos XML API Element for show interface terse */
    var $get-interface-rpc = <get-interface-information> {
                <terse>;
              }

    /* Retrieve API results through jcs:invoke */
    var $results = jcs:invoke( $get-interface-rpc );

    /* Use the command-line argument if provided otherwise ask user through
jcs:get-input() */
    var $interface-value = {
                if( string-length( $interface ) > 0 ) {
                  expr $interface;
                }
                else {
                  expr jcs:get-input("Enter interface: ");
                }
              }

    /* Locate the matching node and assign to $admin-status */
    var $admin-status = $results/physical-interface[name==$interface-
value]/admin-status;

    /* Output the interface and status to the console */
    <output> "The admin status of " _ $interface-value _ " is " _ $admin-
status;
  }
}
```

The difference between the new script and the original version is the addition of the `$interface-value` conditionally-set variable. If the `$interface` parameter has been entered through the command-line (as indicated by having a string-length greater than `0`) then its entered value is used. Otherwise, the `jcs:get-input()` function is invoked with a prompt of "`Enter interface:`". The user types in the interface name and presses enter, after which the inputted string is assigned to the `$interface-value` variable. This new variable is now used by the following lines to correctly retrieve the interface admin status from the `$results` variable and to print the interface name to the console.

```
user@Junos> op show-admin-status interface fxp0
The admin status of fxp0 is up

user@Junos> op show-admin-status
Enter interface: fxp0
The admin status of fxp0 is up
```

TIP The result tree is only processed after a script is executed, but the `jcs:get-input()` and `jcs:get-secret()` functions run as part of the script processing. This might cause your output to appear in the incorrect order because any use of the `<output>` result tree element is only displayed after the script terminates. An alternative method of writing to the console, which occurs during script processing, is to use the `jcs:output()` function. Here is an example :note that it must always be preceded by the `expr` statement: `expr jcs:output("Hello World!");`

jcs:get-secret()

The `jcs:get-secret()` function works in the same way as `jcs:get-input()` except that the user input is not echoed to the screen. This makes the function ideal when the user must enter sensitive information such as passwords.

ALERT! jcs:get-secret() requires Junos 9.6 or later.

Try It Yourself: Interacting with the User

Modify your script displaying the MTU of a single physical interface. Add a check to see if the command-line argument for the interface has been entered. If it has not, then request the information from the user through the jcs:get-input() function.

Writing to the Syslog

Junos scripts can write messages to the syslog by using the `jcs:syslog()` function. This requires two arguments. The first is a string that defines the facility and severity at which the message should be logged. The second is the message string to be logged.

A single string expresses the facility and severity with a period inbetween, for example "external.error" or "daemon.info". The available facilities and severities are listed in Tables 4.3 and 4.4.

Table 4.3 Syslog Facilities

Facility String	Description
auth	Authorization system
change	Configuration change log
conflict	Configuration conflict log
daemon	Various system processes
external	Local external applications
firewall	Firewall filtering system
ftp	FTP process
interact	Commands executed via CLI
pfe	Packet forwarding engine
user	User processes

Table 4.4 Syslog Severity Levels

Severity	Description
alert	Conditions that require immediate correction
crit	Critical conditions
debug	Debug messages
emerg / panic	Panic conditions
err /error	Error conditions
info	Informational messages
notice	Non-error conditions that require special handling
warn / warning	Warning messages

When including the `jcs:syslog()` function in a script it must be preceded by the `expr` statement. While there is no result returned to write to the result tree, SLAX requires that scripts try to do something with function results, even if they are always blank:

```
expr jcs:syslog("external.warn", "The script changed the configuration" );
```

The above code would result in the following being logged to the syslog:

```
May 21 21:45:13 Junos cscript: %EXTERNAL-4: The script changed the configuration
```

Syslog messages from Junos scripts always begin with `cscript:` .
The %EXTERNAL-4 is present because explicit-priority has been activated in the configuration for the syslog file. It shows that the message was logged by the correct facility at the desired severity.

Here is an example script that allows any desired syslog message to be logged from the CLI. The facility, severity, and message are all entered through command-line arguments:

```
/* log-message.slax */
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";

/* This is imported into the Junos CLI help text */
var $arguments = {
  <argument> {
    <name> "facility";
    <description> "Facility for the syslog message";
  }
  <argument> {
    <name> "severity";
    <description> "Severity level of the syslog message";
  }
  <argument> {
    <name> "message";
    <description> "Message to send to syslog";
  }
}

/* Command-line arguments */
param $facility;
param $severity;
param $message;

match / {
  <op-script-results> {

/* Assemble the facility-severity string */
    var $facility-severity = $facility _ "." _ $severity;

/* Log message to syslog */
    expr jcs:syslog( $facility-severity, $message );
  }
}
```

If the log-message.slax script is run with the following arguments:

```
user@Junos> op log-message facility user severity notice message "Test
Message"
```

Then it logs this to the syslog (with explicit-priority set):

```
May 21 21:50:18 Junos cscript: %USER-5: Test Message
```

NOTE `jcs:syslog()` requires Junos 9.0 or later.

Try It Yourself: Writing to the Syslog

Create an op script that logs the user name, script, product, and hostname to the syslog from the user facility with a severity level of info.

Reading the Configuration

Junos scripts can retrieve the current configuration by sending the <get-configuration> API Element to jcs:invoke():

```
var $configuration = jcs:invoke( "get-configuration" );
```

There are a number of attributes available for <get-configuration>, the most useful of which is database. It can be set to either committed or candidate, and it indicates which configuration database should be returned. The candidate database is returned by default if the database attribute is missing.

```
var $rpc = <get-configuration database="committed">;
var $committed-configuration = jcs:invoke( $rpc );
```

MORE? Additional attributes is used with the <get-configuration> API Element. Consult the Junoscript API Guide in the Junos documentation at www.juniper.net/techpubs/ for more information.

The entire configuration is returned in XML format enclosed within a parent <configuration> element. To see the XML structure of the configuration on a Junos device, append | display xml to the show configuration command:

```
user@Junos> show configuration | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.6I0/junos">
  <configuration ...attributes were cut... >
     <version>9.6I0 [builder]</version>
     <groups>
       <name>re0</name>
       <system>
         <host-name>Junos</host-name>
         <time-zone>America/Los_Angeles</time-zone>
     <snip>
```

The <get-configuration> API Element does not have to return the entire configuration. When the script requires only portions of the configuration then specify those hierarchies within <get-configuration>:

```
var $rpc = <get-configuration database="committed"> {
       <configuration> {
         <protocols> {
           <bgp>;
         }
         <policy-options>;
       }
     }
```

As shown above, to request a subset of the configuration, enclose the desired hierarchies within a <configuration> child element of the <get-configuration> API Element. The example above only retrieves the bgp and policy-options hierarchy levels.

Configuration settings can be retrieved through location paths in the same way as operational results. This example shows how to do it:

```
/* show-name-servers.slax */
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> {

    /* Junos XML API Element to retrieve the configuration */
    var $config-rpc =  <get-configuration> {
              <configuration> {
                <system>;
              }
            }

    /* Request configuration and assign to $config variable */
    var $config = jcs:invoke( $config-rpc );

    /* Extract the name-servers from the config and assign to variable */
    var $name-servers = $config/system/name-server;

    /*
     * If no name-servers are present then output message, otherwise
     * output all the name-server names/addresses.
     */
    if( jcs:empty( $name-servers ) ) {
      <output> "There are no name servers defined.";
    }
    else {
      <output> "Here are the name-servers:";
      for-each( $name-servers ) {
        <output> ./name;
      }
    }
  }
}
```

The script begins by requesting the current configuration of the system hierarchy. The `$config` variable has the `<configuration>` element node as its reference point, so the first step of the location path is the main hierarchy level (in the example: `system`). The script then loads all the `<name-server>` element nodes from the system hierarchy into a node-set variable `$name-servers`. What happens next depends on the value of `$name-servers`. The `jcs:empty()` function returns true if its node-set argument is empty, otherwise it will return false. In the script code, if the `$name-servers` variable is empty, so `jcs:empty()` returns true, then the output string will express that there are no name-servers. Otherwise, if name-servers are present in the configuration then a `for-each` statement will loop through every element node within the `$name-servers` variable and output the address of the name-server. Here is the output:

```
user@Junos> op show-name-servers
Here are the name-servers:
192.168.16.10
192.168.35.10
```

This second example looks at the `routing-options` hierarchy level. It checks to see if an autonomous-system number has been configured and reports if one is present or not. It also reports all static routes that have been configured:

```
/* show-routing-options.slax */
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> {

    /* API Element to retrieve the routing-options configuration in XML */
    var $config-rpc =  <get-configuration> {
              <configuration> {
                <routing-options>;
              }
            }

    /* Send API Element to Junos via jcs:invoke and retrieve config in XML
format */
    var $config = jcs:invoke( $config-rpc );

    /* Extract the ASN from the configuration using a location path */
    var $asn = $config/routing-options/autonomous-system/as-number;

    /* If the ASN is defined then output it to the console */
    if( jcs:empty( $asn ) ) {
      <output> "There is no ASN defined.";
    }
    else {
      <output> "The ASN is: " _ $asn;
    }

    /* Extract all the static routes from the configuration using a location
path */
    var $static-routes = $config/routing-options/static/route;

    /* If there are static routes present then loop through and display them
*/
    if( jcs:empty( $static-routes ) ) {
      <output> "There are no static routes.";
    }
    else {
      for-each( $static-routes ) {
        <output> "There is a static route to " _ name _ " with next-hop " _
next-hop;
      }
    }
  }
}
```

If the `routing-options` configuration looks like this:

```
<configuration>
    <routing-options>
      <static>
        <route>
          <name>10.1.0.0/16</name>
          <next-hop>192.168.1.1</next-hop>
        </route>
        <route>
          <name>10.2.0.0/16</name>
          <next-hop>192.168.1.1</next-hop>
        </route>
      </static>
      <autonomous-system>
        <as-number>65500</as-number>
      </autonomous-system>
    </routing-options>
</configuration>
```

Then this will be the output from the script:

```
user@Junos> op show-routing-options
The ASN is: 65500
There is a static route to 10.1.0.0/16 with next-hop 192.168.1.1
There is a static route to 10.2.0.0/16 with next-hop 192.168.1.1
```

Try It Yourself: Reading the Junos Configuration

Create an op script that reads the configuration and outputs all the syslog file names to the console.

Changing the Configuration

Junos scripts are capable of changing, as well as reading, the configuration. When used in this manner op scripts can perform structured configuration changes, allowing users with less Junos familiarity to safely alter the configuration. As an example, consider a provisioning script that could ask relevant questions for a new connection and then automatically commit the needed configuration. This decreases the complexity in adding new customers even when sophisticated policies are in place, and it guarantees that all customers are added to the configuration using the established guidelines.

To change the configuration use the `jcs:load-configuration` template. This template is included in the junos.xsl default import file and is available for use by all op scripts (in Junos 9.3 and later). Before using `jcs:load-configuration` a connection must be opened. This is needed because multiple steps must occur for a configuration change to be successful:

- The configuration database is locked.
- The configuration changes are loaded.
- The configuration is committed.
- The configuration database is unlocked.

All of these actions must be performed in a sequence with no interference from other users or scripts. Junos uses connections to ensure that the steps are performed together rather than in isolation. These connections are first created by the `jcs:open()` function, which returns a connection identifier. When the script is ready to make a configuration change it passes this connection to the `jcs:load-configuration` template. After the changes have been completed, the connection should be closed by using the `jcs:close()` function.

NOTE The `jcs:open()` and `jcs:close()` functions and the `jcs:load-configuration` template are only available in Junos 9.3 and beyond.

Configuration changes made with the `jcs:load-configuration` template are made in exclusive configuration mode. If any users are currently editing the configuration or if the database is currently locked then the attempted changes fail. The `jcs:load-configuration` template first locks the database; it then loads the configuration change, performs a commit, and unlocks the database.

The script must provide the connection, and the configuration changes the `jcs:load-configuration` template through its parameters. The change is expressed in SLAX

abbreviated XML format and is enclosed within a <configuration> element. The entire hierarchy level must be included for the change, which is merged into the existing configuration:

```
<configuration> {
  <routing-options> {
    <static> {
      <route> {
        <name> "10.3.0.0/16";
        <next-hop> "192.168.1.1";
      }
    }
  }
}
```

The above XML structure could be used to add a new static route for 10.3.0.0/16.

The template parameters used by jcs:load-configuration are $connection for the connection and $configuration for the configuration change. An example of the steps needed to make a configuration change can be seen in the following:

```
/* The configuration change must be defined */
var $configuration-change = <configuration> {
              <routing-options> {
                <static> {
                   <route> {
                     <name> "10.3.0.0/16";
                     <next-hop> "192.168.1.1";
                   }
                }
              }
            }

/* A connection must be opened */
var $connection = jcs:open();

/*
 * The connection and change are set as parameters to the jcs:load-
configuration
 * template which performs the change. The := operator is used to ensure that
the
 * $results variable is a node-set rather than a result tree fragment.
 */
var $results := { call jcs:load-configuration( $connection, $configuration
= $configuration-change ); }

/* Check for errors - report them if they occurred */
if( $results//xnm:error ) {
  for-each( $results//xnm:error ) {
    <output> message;
  }
}

/* The connection is closed */
var $close-results = jcs:close($connection);
```

NOTE The location path used to check for errors from templates like jcs:load-configuration is different than the location path for errors from functions like jcs:invoke(). With jcs:load-configuration the location path is: "$results//xnm:error". With jcs:invoke() the location path is "$results/..//xnm:error". (Both of these examples assume that the variable name used is $results.)

MORE? Find more examples of op scripts – including a script for configuring static routes, a script that extracts a policy chain, a script that lets user self-serve their local account by changing their password, and a script to change the default behavior of standard operational mode commands – in the Appendices of this book.

Part Two

Applying Junos Event Automation

Chapter 5

Introducing Event Scripts

The Junos automation toolset is a standard part of the Junos operating system available on all Junos platforms including routers, switches, and security devices. This Part Two continues teaching the core concepts of Junos automation begun in the first part, *Applying Junos Operations Automation*. It explains the SLAX scripting language and describes how to use op scripts, which are one type of Junos automation script. Here, we describe how to configure automatic responses to system events through event policies and event scripts.

Junos Automation Overview

Junos automation enables an organization to embed its wealth of knowledge and experience of operations directly into Junos devices:

- Business rules automation: Compliance checks can be enforced. Change management helps avert human error.
- Provisioning automation: Abstracts and simplifies complex configurations. Automatically corrects errors.
- Operations automation: Creates customized commands and outputs to streamline tasks and ease troubleshooting.
- Event Automation: Pre-defines responses for events, allowing the Junos device to monitor itself and react as desired.

Through automation, Junos empowers network operators to scale by simplifying complex tasks, maximizing uptime, and optimizing operational efficiency.

Event Scripts

Part One, *Applying Junos Operations Automation*, discussed using op scripts. An op script is a customized command that can be executed from the CLI prompt in the same way as a standard Junos command. Op scripts are used to gather and display desired information, to perform controlled configuration changes, or to execute a group of operational commands. But Junos offers a rich set of automation capabilities beyond what op scripts alone can provide. Op scripts are run manually through invocation at the CLI prompt, or upon user login; they are designed to be interactive and not suited to react automatically to a system event.

This second part of Junos automation, the ability to react to system events, is the realm of event scripts and event policies. Junos automatically executes event scripts in response to a system event. Valid events, which can result in an automatic reaction, include syslog messages, SNMP traps, chassis alarms, and internal timers.

Although op scripts execute ondemand in response to a command, and event scripts automatically execute in response to an event, the two types of scripts otherwise share many similarities. Both types of scripts are written using the same scripting language (XSLT or SLAX), and both follow the same programming rules and guidelines. In fact, at a high level it is appropriate to think of an event script as simply an op script that is automatically executed by the Junos device instead of being manually invoked by a user.

Chapter 8 discusses the unique characteristics and capabilities available to event scripts, but many scripts never need to take advantage of this extra functionality. And for these basic scripts, the code looks essentially the same for both script types. It is only the manner in which the scripts are executed – on demand versus automatic – that distinguishes them as either an op script or an event script.

Event Script Examples

Event scripts can automate a response to almost any event in the Junos device, and so have countless applications, for example:

- In response to a protocol fault, an event script could gather trouble- shooting information from the local Junos device as well as its protocol peers. The device could store this information locally or automatically transmit it to a remote server for storage and analysis.
- In response to the time-of-day, an event script could perform configuration changes, such as automatically enabling specific firewall filter terms for night time use only.
- An event script could perform a comprehensive daily configuration audit to verify the active configuration against a remotely stored baseline. Execution of the script could correct violations automatically, or generate a syslog message to report the problem.
- At hourly intervals, an event script could inspect the Junos device for ongoing issues requiring operation attention such as core dump files or active chassis alarms. Appropriate syslog messages could be logged, reminding operational staff that they must login and troubleshoot the problem.

MORE? To see more script examples go to the online script library at www.juniper.net/scriptlibrary.

Advanced Insight Solutions

Advanced Insight Solutions (AIS) services from Juniper Networks are a good example of the possibilities that event scripts provide to embed troubleshooting intelligence into Junos devices. AIS is a support automation platform for Junos that streamlines the detection, isolation, and resolution of network faults and incidents. The Advanced Insight event scripts are a key component of AIS. These event scripts handle over 300 system events automatically, according to the recommended best practices of the Juniper Networks Technical Assistance Center (JTAC). When a problem occurs, the event scripts automatically gather all the needed troubleshooting information and upload it to a destination server. This information can then be automatically passed on to JTAC for case creation and further assistance. Additionally, AIS uses other scripts to proactively monitor the Junos devices, looking for potential problems.

MORE? For more information on AIS see this webpage: www.juniper.net/techpubs/software/management/ais/.

Event Policies

Event scripts are not self-executing, they are automatically triggered by event policies. These policies are part of the Junos configuration; they instruct the event processing daemon to perform specific actions in response to system events. Executing an event script is one of the possible event policy actions in addition to being the focus of this part. Chapters 6 and 7 discuss event policies in depth.

NOTE The Junos operating system divides its functionality into multiple software processes, known as daemons. This modularization improves the overall reliability of Junos, as any failures are contained within a single process rather than affecting the entire operating system. The event processing daemon is named `eventd`; it is responsible for handling all system events. This task includes relaying syslog messages as well as triggering event policies.

Configuration / Storage

For an event policy to execute an event script, the script must be stored on the Junos device and enabled in the device configuration. There are two ways to store and enable event scripts; the method used depends on the Junos software version on the device. In both cases only super-users, users with the `all` permission bit, or users that have been given the `maintenance` permission bit are permitted to enable or disable Junos scripts in the configuration.

MORE? For more information on configuration permissions, see the *System Basics* manual within the Junos documentation at www.juniper.net/techpubs.

TIP Devices with multiple routing-engines must have the script file copied into the event script directory on all routing-engines. The script must also be enabled within the configuration of each routing-engine. Typically the configuration on the other routing-engine is done automatically through configuration synchronization, but if the configurations are not synchronized between routing-engines then the script must be manually enabled on both routing-engines manually.

Before Junos 9.0

Prior to Junos 9.0, event scripts were stored and enabled in the same manner as op scripts. That is, they were saved in the `/var/db/scripts/op` directory and enabled under the `system scripts op` hierarchy level. For example:

```
set system scripts op file example-event-script.slax
```

When using this method, no difference exists in how the administrator stores and configures an op script and an event script. Users can invoke an event script set up like this from the command line in the same way as a op script. This is because these event scripts are technically op scripts executed by an event policy, rather than true event scripts. But, because they execute automatically, they are still referred to as event scripts, although they lack some of the event script capabilities discussed in Chapter 8.

Junos 9.0 and Beyond

Beginning in Junos 9.0, the Juniper engineering staff added new capabilities to event scripts that further differentiate them from op scripts. Because of the new functionality it was necessary to more clearly differentiate between op scripts and event scripts through a different storage location and enabling configuration statement. Table 5.1 summarizes these changes.

In Junos 9.0 the storage location of event scripts changed to `/var/db/scripts/event`, and the configuration to enable them was added under the `event-options event-script` hierarchy. For example:

```
set event-options event-script file example-event-script.slax
```

While you can still use the old storage and configuration method for event scripts, Juniper recommends that you follow the new approach on all devices using Junos 9.0 or later versions. This new method gives event scripts access to specific capabilities discussed in Chapter 8:

- Embedded event policies
- The <event-script-input> XML data

NOTE The <event-script-input> XML data is provided to event scripts starting with Junos 9.3.

Table 5.1 Event Script Storage and Enablement

	Before Junos 9.0	Junos 9.0 and Beyond
Storage Location	/var/db/scripts/op	/var/db/scripts/event
Configuration Location	system scripts op	event-options event-script

BEST PRACTICE Use the new methods to store and enable event scripts unless a Junos version prior to 9.0 is in use.

NOTE All event policy configuration examples in this book also include the configuration necessary to enable the event script under event-options event-script. This statement should be ignored if the event script is being enabled under system scripts op.

Event Script Execution Message

Starting in Junos 9.6, Junos logs a message to the syslog when it executes an event script. This syslog message indicates if the event script is being read from /var/db/scripts/event (and is a true event script) or from /var/db/scripts/op.

Here are examples of the logged message:

- When executed from /var/db/scripts/op:

```
Aug 19 15:18:55  Junos eventd[949]: EVENTD_ESCRIPT_EXECUTION: Trying to execute the script 'log-
message.slax' from '/var/db/scripts/op/'
```

- When executed from /var/db/scripts/event:

```
Aug 19 15:23:08  Junos eventd[949]: EVENTD_ESCRIPT_EXECUTION: Trying to execute the script 'log-
message.slax' from '/var/db/scripts/event/'
```

Event Script Boilerplate

When writing Junos scripts, it is best to always work from the standard boilerplate. This greatly simplifies script writing, as there is no need to memorize the necessary name-space URLs. Instead, just copy and paste the boilerplate and add your script code within it. The boilerplate used for writing event scripts is almost identical to the boilerplate used for op scripts. The only difference is the top-level result tree element, which is included in the boilerplate. Op scripts use <op-script-results> for this top-level element, but event scripts use <event-script-results>. Here is the boilerplate to use when writing event scripts:

```
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 / {
    <event-script-results> {

     /* Your script code goes here */

    }
}
```

version: while version 1.0 is currently the only available version of the SLAX language, the version line is required at the beginning of all Junos scripts.

ns: a ns statement defines a namespace prefix and its associated namespace URL. The following three namespaces must be included in all Junos scripts:

- 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";

It is easiest to just copy and paste these namespaces into each new script as part of the boilerplate rather than trying to type them out by hand.

import: the import statement is used to import code from one script into the current script. As the `junos.xsl` script contains useful default templates and parameters, all scripts should import this file. The `import "../import/junos.xsl";` line from the boilerplate is all a script needs to accomplish this.

match /: this code block is the main template of the event script. In the standard boilerplate it includes the `<event-script-results>` result tree element:

```
match / {
    <event-script-results> {

     /* Your script code goes here */

    }
}
```

The boilerplate includes the `<event-script-results>` element to simplify writing event scripts. You can include SLAX statements within the `<event-script-results>` code block without interfering with the created result tree. The script processor can differentiate between statements that should be executed and XML elements that should be added to the tree. The element `<event-script-results>` should always be the top-level element in an event script result tree. This element indicates to Junos that the result-tree instructions originated from an event script. There is no action performed by the `<event-script-results>` element, it simply contains the child elements. The child elements of `<event-script-results>` provide instructions that Junos processes after the script has terminated.

NOTE Event scripts enabled under `system scripts op` should follow the op script boilerplate and use the `<op-script-results>` top-level result tree element instead of the `<event-script-results>` element.

Chapter 6

Configuring Event Policies

Junos automation scripts can automate many operation steps in Junos. Event policies and event scripts work together to automate how Junos responds to system events. The event script contains the instructions for Junos to run, but it is the event policy – specifically the event processing daemon – that informs Junos when to execute the event script. This chapter begins by discussing what events are and how to identify them. It then provides an overview of event polices and their configuration.

Events Overview

In the Junos operating system an event can be a syslog message, a SNMP trap, a chassis alarm, or an internal time-based trigger. Event examples include:

- Configuration commits
- Interfaces going up or down
- New hardware components being inserted or removed
- Protocol adjacency changes

Most events have specific event IDs, which are string names that can be seen in the syslog file:

```
Jul 22 10:19:52  Junos mgd[3578]: UI_DBASE_LOGIN_EVENT: User 'user'' entering
configuration mode
Jul 22 10:19:53  Junos mgd[3578]: UI_COMMIT: User 'user' requested 'commit'
operation (comment: none)
Jul 22 10:19:56  Junos mgd[3578]: UI_DBASE_LOGOUT_EVENT: User 'user' exiting
configuration mode
```

Each line above shows a system event. The three events are:

- `UI_DBASE_LOGIN_EVENT`: A user-entered configuration mode
- `UI_COMMIT`: A configuration commit was performed
- `UI_DBASE_LOGOUT_EVENT`: A user exited configuration mode

The event ID appears in uppercase following the daemon name (`mgd` in the above example) and process ID if applicable. A message typically follows the event ID, this message provides more explanation of the event that occurred.

BEST PRACTICE Syslog messages from the event processing daemon (eventd) are not treated as system events. They should never be included in an event policy configuration.

NOTE As discussed in Chapter 5, Junos is modularized into multiple software processes known as daemons. Some of the commonly known Junos daemons include the routing daemon (`rpd`), the management daemon (`mgd`), the chassis daemon (`chassisd`), and the event processing daemon (`eventd`).

Event Attributes

Events contain attributes that provide additional information to describe the event. For example, the three events shown in the prior section each contain an attribute of `username`. This attribute indicates which user account entered configuration mode, performed the commit, and then left configuration mode. The extra details provided by attributes allow event policies to base their reaction not only on the event ID itself, but also on specific characteristics of the event. In the event examples of the prior

section, event policies could react differently based on what user performed the commit or entered and exited configuration mode.

Identifying Events and Attributes

Syslog files contain a record of the events that have already occurred on Junos devices. Analyzing these syslog files reveals the event ID that corresponds to each of these past events. For a complete view, consult the `help syslog` Junos CLI command. It displays all the potential system events in the Junos device to which an event policy can react:

```
user@Junos> help syslog
Syslog tag                        Help
ACCT_ACCOUNTING_FERROR            Error occurred during file processing
ACCT_ACCOUNTING_FOPEN_ERROR       Open operation failed on file
ACCT_ACCOUNTING_SMALL_FILE_SIZE Maximum file size is smaller than record size
ACCT_BAD_RECORD_FORMAT            Record format does not match accounting profile
ACCT_CU_RTSLIB_ERROR              Error occurred obtaining current class usage statistics
ACCT_FORK_ERR                     Could not create child process
ACCT_FORK_LIMIT_EXCEEDED          Could not create child process because of limit
...
```

As shown above, executing the `help syslog` command without any arguments shows the complete listing of event IDs along with their brief description. Including a specific event ID as an option to the `help syslog` command shows the full description of the event:

```
user@Junos> help syslog UI_COMMIT
Name:        UI_COMMIT
Message:      User '<username>' requested '<command>' operation (comment: <message>)
Help:        User requested commit of candidate configuration
Description:   The indicated user requested the indicated type of commit operation on the candidate
configuration and added the indicated comment. The 'commit' operation applies to the local Routing
Engine and the 'commit synchronize' operation to both Routing Engines.
Type:        Event: This message reports an event, not an error
Severity:     notice
```

This output includes the event ID (the name of the event), message format, syslog severity, and other descriptive fields. Additionally, the output includes event attributes contained in the message string, where each attribute is enclosed within < >. The above output shows that the `UI_COMMIT` event has the following attributes:

- `username`
- `command`
- `message`

The same method can be used to learn the attributes of the `UI_DBASE_LOGIN_EVENT` event or any other system event:

```
user@Junos> help syslog UI_DBASE_LOGIN_EVENT
Name:        UI_DBASE_LOGIN_EVENT
Message:      User '<username>' entering configuration mode
Help:        User entered configuration mode
Description:   The indicated user entered configuration mode (logged into the configuration
database).
Type:        Event: This message reports an event, not an error
Severity:     notice
```

As the output shows, this event only has a single attribute:

- `username`

Syslog Structured-Data

Another way to learn what attributes correspond to an event is to enable `structured-data` for a syslog file. When this statement is included in the syslog configuration the Junos device uses a more verbose format to log its syslog messages:

```
system {
   syslog {
      file syslog {
         any notice;
         structured-data;
      }
   }
}
```

Here are the same three events that were shown initially, but now the syslog file uses the structured-data format, which causes more information to be included in each syslog message:

```
<189>1 2009-07-22T10:41:38.611-07:00 Junos mgd 3578 UI_DBASE_LOGIN_EVENT [junos@2636.1.1.1.2.2
username="roy"] User 'roy' entering configuration mode

<189>1 2009-07-22T10:41:40.645-07:00 Junos mgd 3578 UI_COMMIT [junos@2636.1.1.1.2.2 username="roy"
command="commit" message="none"] User 'roy' requested 'commit' operation (comment: none)

<189>1 2009-07-22T10:41:44.041-07:00 Junos mgd 3578 UI_DBASE_LOGOUT_EVENT [junos@2636.1.1.1.2.2
username="roy"] User 'roy' exiting configuration mode
```

This more verbose format has the advantage of showing not only the event attributes but also the attribute values. The attribute names and values follow the event ID and are all enclosed within square brackets. In the example above, the following information can be learned by examining the event attributes:

- The user `"roy"` entered configuration mode.
- The user `"roy"` committed the configuration using the `"commit"` command with no message included.
- The user `"roy"` exited configuration mode.

Nonstandard Events

Not all events have an assigned event ID. Many events are instead logged under generic IDs that identify their origin. Each of these nonstandard events, sometimes called pseudo events, have a single attribute called `message` that contains the syslog message that is logged when the event occurs. When multiple events all share the same generic event ID, event policies need to look for the desired syslog message by checking the value of the message attribute of these events. A later section in this chapter discusses how to configure attribute comparisons within event policies. This method allows Junos devices to react to a specific event, even though that event shares an ID with other system events. The table below identifies the nonstandard event IDs:

Nonstandard Event IDs

Table 6.1 reviews the various event IDs and their descriptions.

Table 6.1 Event ID Descriptions

Event ID	Description
SYSTEM	Messages from Junos daemons and utilities.
KERNEL	Messages from Junos kernel.
PIC	Messages from physical interface cards.
PFE	Messages from the packet forwarding engine.
LCC	Messages from TX Matrix line-card chassis.
SCC	Messages from TX Matrix switch-card chassis.

Here are examples of syslog messages from nonstandard events:

```
Jul 22 11:43:49  Junos /kernel: fxp1: link DOWN 100Mb / full-duplex
Jul 22 11:43:50  Junos /kernel: fxp1: link UP 100Mb / full-duplex
Jul 22 11:47:05  Junos xntpd[5595]: kernel time sync enabled 2001
```

The first two events are from the kernel, which means they are logged using the KERNEL event ID. The third event is logged from a daemon but lacks a specific event ID. The event is instead logged using the SYSTEM event ID. Each of these events can be matched by referring to their message attribute as an upcoming section demonstrates.

Logger

The testing process is an essential part of creating event policies and event scripts, but testing some events can be tricky if they are difficult to manually generate. Luckily, Junos ships with a test utility known as logger that can artificially generate any system event. Logger makes it possible to test event policies and event scripts successfully regardless of whether the desired event is simple or arcane.

NOTE Logger is a Junos shell program that is unsupported and should not be used on production devices. But logger is well suited for use in lab environments where event policies and event scripts are being developed and verified.

ALERT! Logger should be used only in Junos 9.3R4 and later. Unexpected failure may occur if used in a prior version.

How to use the logger test utility

1. The logger utility is a shell command, and so the user must first start a system shell by invoking the start shell command:

```
user@Junos> start shell
%
```

2. The logger utility has the following command syntax:

```
logger -e EVENT_ID -p SYSLOG_PRIORITY -d DAEMON -a ATTRIBUTE=VALUE MESSAGE
```

Only the EVENT_ID is required, and it must be entered entirely in uppercase:

```
% logger -e UI_COMMIT
```

The above command causes a `UI_COMMIT` event to be generated, originated from the logger daemon, with no attributes, no message, and a syslog facility/severity of user/notice.

The default settings can be altered by using one of the optional command line arguments.

3. For an alternate syslog facility/severity use the `-p` argument and specify the facility/severity in the same `facility.severity` format used by the `jcs:syslog()` function:

```
% logger -e UI_COMMIT –p external.info
```

MORE? See *Part One: Applying Junos Operations Automation* for a table that lists the valid syslog facilities and severities for the `jcs:syslog()` function.

4. To alter what daemon generated the event, use the `-d` argument:

```
% logger -e UI_COMMIT –d mgd
```

5. Include attributes for the event by using the `-a` argument. Use the argument multiple times if more than one attribute is needed. The attribute name must be in lowercase and should be followed by an equal sign and the desired value:

```
% logger -e UI_COMMIT -a username=user -a command=commit
```

6. The syslog message follows all the command line arguments. Quotes are not required but are recommended for clarity:

```
% logger -e UI_COMMIT -d mgd "This is a fake commit."
```

The above command causes the following message to be shown in the syslog:

```
Jul 22 12:47:03  Junos mgd: UI_COMMIT: This is a fake commit.
```

NOTE When using the logger utility the event ID must always be in uppercase and the attribute names must always be in lowercase.

Try it Yourself: Simulating Events with the Logger Utility

1. Use `help syslog` to identify an event of interest and its attributes.

2. Configure a syslog file to use structured-data format.

3. Using the logger utility, generate an artificial version of the selected event including values for all of its attributes.

4. Verify that the event was created as expected by viewing the structured-data syslog file.

Event Policy Overview

Event policies are created within the Junos configuration to instruct Junos to perform specific actions in response to system events. Each event policy is an `if-then` construct. The `if` portion of the policy consists of one or more match conditions. If these match conditions are correctly met, then Junos performs the actions specified under the `then` statement. A single event can trigger more than one event policy. The Junos event processing daemon processes the event policies sequentially, in configuration order, and performs the actions of all matching policies in response to the event.

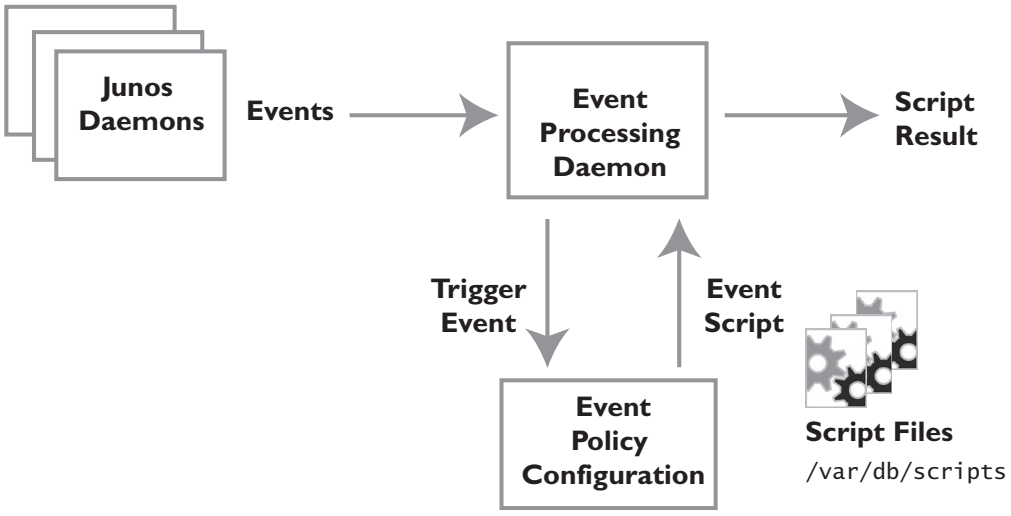


Figure 6.1 The Flow of Event Processing

Event policies are configured under the `event-options` hierarchy level. Each event policy is created with a unique name and contains a matching event ID as well as a `then` statement with instructions to process in reaction to the specific events.

Basic Configuration

The minimum configuration for an event policy is an `events` statement and at least one `then` action:

```
event-options {
    policy example {
        events ui_commit;
        then {
            event-script example-script.slax;
        }
    }
    event-script {
        file example-script.slax;
    }
}
```

NOTE The configuration necessary to enable the event script is also included in this and all other event policy examples. See Chapter 5 for further details.

The example above creates an event policy named `example`. This policy matches on the `UI_COMMIT` event, which is referred to as the trigger event for the policy. Anytime the trigger event occurs, Junos automatically executes the `example-script.slax` event script.

Multiple events can be configured within the same event policy. When multiple events are configured, the policy can be triggered by any one of the events:

```
event-options {
```

```
    policy example {
        events [ ui_dbase_login_event ui_dbase_logout_event ];
        then {
            event-script example-script.slax;
        }
    }
    event-script {
        file example-script.slax;
    }
}
```

The example event policy above executes the `example-script.slax` event script anytime either a `UI_DBASE_LOGIN_EVENT` or `UI_DBASE_LOGOUT_EVENT` event occur.

TIP While the trigger event can be in uppercase or lowercase, lowercase provides the advantage that command completion can be used to auto-complete the event ID.

Maximum Policy Count

Junos can run a maximum of 15 event policies at the same time. This limit is imposed to conserve resources and prevent event looping. If the number of running event policies has reached the limit, Junos does not run further policies and instead writes the following message to the syslog:

```
EVENTD_POLICY_LIMIT_EXCEEDED: Unable to execute policy <name> because
current number of policies (15) exceeds system limit (15)
```

Your First Event Script

Here is an example of a basic event script called `log-hello-world.slax`. When executed, this script logs the message "Hello World!" to the syslog:

```
/* log-hello-world.slax */
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 / {
    <event-script-results> {

        /* Send Hello World! to syslog from facility external with severity
info */
        expr jcs:syslog("external.info", "Hello World!");

    }
}
```

The next step is to create the needed event policy to trigger Junos to execute the event script in response to the desired event. Because the example event script is very generic one could configure Junos to execute it following any of the hundreds of available events, but in this example the `UI_COMMIT` event is the trigger event:

```
event-options {
    policy hello-world {
        events ui_commit;
```

```
        then {
            event-script log-hello-world.slax;
        }
    }
    event-script {
        file log-hello-world.slax;
    }
}
```

With the script correctly loaded and the above event policy configured, the Junos device writes "Hello World!" to the syslog every time a commit is issued:

```
[edit]
user@Junos# commit
commit complete

user@Junos> show log messages | match cscript
Jul  8 14:35:34  Junos cscript: %EXTERNAL-6: Hello World!
```

MORE? For more information on the usage of the `jcs:syslog()` function refer to the *Part One: Applying Junos Operations Automation* book. The function can be used the same way in `op`, `event`, or `commit` scripts.

Try it Yourself: Logging a Syslog Message in Response to an Event

1. Using the `log-hello-world.slax` script as an example, create an event script that logs a message to the syslog.

2. Copy the event script to the Junos device and enable the script in the configuration.

3. Select an event ID of interest and configure an event policy that executes the event script in response to the system event.

4. Use the logger utility to simulate the event and verify that the desired message is logged to the syslog.

Correlating Events

While it is often possible to construct event policies triggered by only a single event, other event policies should only respond when multiple events occur in close relation to each other.

As an example, assume that an organization wishes to record all failed configurations that resulted in a loss of connectivity to their management network. They have a RPM (Real-Time Performance Monitoring) test setup that verifies the reachability of the management network every minute, and their operators always commit changes using the `commit confirmed` CLI command (allowing the configuration to automatically rollback if the configuration change impacts connectivity). The goal of the automation is for the Junos device to copy the configuration overridden by the automatic rollback to a specific location for later examination. This goal can be accomplished by using an event policy and event script.

MORE? For more information on Real-Time Performance Monitoring configuration see the *Services Interfaces* manual within the Junos documentation at www.juniper.net/techpubs/.

MORE? For more information on the `commit confirmed` command see the *CLI User Guide* within the Junos documentation at www.juniper.net/techpubs/.

The event script is named save-rollback.slax. This event script copies the /config/juniper.conf.1.gz rollback file to the /var/tmp directory for permanent storage of the failed configuration.

The event that occurs when a commit confirmed is automatically rolled back is: UI_COMMIT_NOT_CONFIRMED. For learning purposes, this book considers multiple event policy configurations that can be attempted when trying to meet the organization's automation goal. The initial attempts are shown so that their flaws can be discussed before the final configuration is demonstrated.

A first attempt might look like the following:

```
policy faulty-rollback {
   events ui_commit_not_confirmed;
   then {
      event-script save-rollback.slax;
   }
}
```

But there is a problem with the above policy: the policy is triggered every time a UI_COMMIT_NOT_CONFIRMED occurs. This might have happened due to a loss of connectivity, but it also might have happened for a different reason. Perhaps the confirmation was overlooked by the operator, for example. No matter the reason, the UI_COMMIT_NOT_CONFIRMED event by itself cannot communicate why the commit was not confirmed. That information needs to be determined through other sources.

In our current scenario, the best source for this information is to check if a PING_TEST_FAILED event has occurred recently. A PING_TEST_FAILED event occurs every time the RPM test to the management network fails. What is needed is a way to correlate the UI_COMMIT_NOT_CONFIRMED event with the PING_TEST_FAILED event. Doing this would show that both a commit went unconfirmed, and that the management network connectivity was lost at the same time.

This correlation between multiple events is accomplished by using the within statement. This statement consists of a time limit (in seconds) and one or more correlating events. The Junos event daemon has to have received one of the listed correlating events within the given time limit, or Junos does not execute the event policy.

Here is the second attempt at creating the needed event policy:

```
policy faulty-rollback {
   events ui_commit_not_confirmed;
   within 60 events ping_test_failed;
   then {
      event-script save-rollback.slax;
   }
}
```

The trigger event of this policy is UI_COMMIT_NOT_CONFIRMED and the policy has a correlating event of PING_TEST_FAILED, so the event policy is only activated when a UI_COMMIT_NOT_CONFIRMED event occurs if a PING_TEST_FAILED event has occurred within the past 60 seconds. This logic is better than the first attempt as the event policy only runs when connectivity has actually been lost, but one problem still remains with the policy. When a commit confirmed rollback happens, the UI_COMMIT_NOT_CONFIRMED event actually occurs twice. The first time is an announcement that the rollback is going to be performed, and the second time is an indication that the rollback is complete. Here are examples of the two syslog messages that mark these events:

```
UI_COMMIT_NOT_CONFIRMED: Commit was not confirmed; automatic rollback in process
UI_COMMIT_NOT_CONFIRMED: Commit was not confirmed; automatic rollback complete
```

Based on the configuration above, the event policy would execute twice, once when the rollback was starting, and once when the rollback was complete. And a further complication is that the first execution of the script would occur prior to the rollback, while the `juniper.conf.1.gz` file still contains the previous configuration instead of the bad configuration, so the wrong configuration file would be archived. Because of this, it is necessary to ensure that the event policy is only executed for the second `UI_COMMIT_NOT_CONFIRMED` event, and not for the first. This can be done by matching other aspects of the event such as attributes or the syslog message logged by the event. These possibilities are covered in upcoming sections of this book. But for now let's highlight how to achieve the needed policy logic while using only the event ID.

The event policy should only run when both a `PING_TEST_FAILED` event and a `UI_COMMIT_NOT_CONFIRMED` event have occurred recently. Here is the third attempt at creating the needed event policy:

```
policy faulty-rollback {
   events ui_commit_not_confirmed;
   within 60 events [ ping_test_failed ui_commit_not_confirmed];
   then {
      event-script save-rollback.slax;
   }
}
```

This event policy is configured to run only when either a `PING_TEST_FAILED` event or a `UI_COMMIT_NOT_CONFIRMED` event have occurred in the last 60 seconds, and that is the flaw in the configuration: there is an OR relationship between multiple events configured in the same `within` statement.

The event policy should only run when both the `PING_TEST_FAILED` and the `UI_COMMIT_NOT_CONFIRMED` events have occurred recently, not just one or the other. The correct way to do this is with two separate `within` statements. An AND relationship exists between multiple `within` statements, and every `within` statement must have one matching event that occurred within its respective time frame in order for the event policy to run.

The times of each `within` statement must vary, in this case 50 seconds is used as the time-frame in which a `UI_COMMIT_NOT_CONFIRMED` must have been received, which differentiates it from the 60 seconds allowed for the `PING_TEST_FAILED` event.

Now, having gone through all the faulty event policies, the final event policy and event script are displayed below:

```
event-options {
   policy faulty-rollback {
      events ui_commit_not_confirmed;
      within 60 events ping_test_failed;
      within 50 events ui_commit_not_confirmed;
      then {
         event-script save-rollback.slax;
      }
   }
   event-script {
      file save-rollback.slax;
   }
}
```

Here is the `save-rollback.slax` event script that is executed as a result of this event policy:

```
/* save-rollback.slax */
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 / {

   /* Get system time with hyphens instead of spaces */
   var $modified-time = translate( $localtime, " ", "-" );

   /* Use time to differentiate saved rollback versions */
   var $filename = "saved-rollback-" _ $modified-time _ ".gz";

   /*
    * Copy the file using the <file-copy> API Element. This is the XML
    * API form of the "file copy" CLI command.
    */
   var $file-copy-rpc = <file-copy> {
                        <source> "/config/juniper.conf.1.gz";
                        <destination> "/var/tmp/" _ $filename;
                    }
   var $results = jcs:invoke( $file-copy-rpc );

   /* Report any errors or success */
   if( $results/..//xnm:error ) {
      for-each( $results/..//xnm:error ) {
         expr jcs:syslog( "external.error", "File Copy Error: ", message );
      }
   }
   else {
      expr jcs:syslog( "external.info", "Faulty rollback configuration
saved." );
   }
}
```

With the above configuration, the event policy executes the `save-rollback.slax` script in response to a `UI_COMMIT_NOT_CONFIRMED` event when both a `PING_TEST_FAILED` event has occurred in the last 60 seconds and a `UI_COMMIT_NOT_CONFIRMED` event has occurred in the last 50 seconds.

Within NOT Events

The prior section showed how to run an event policy if a correlating event occurred prior to the trigger event, but sometimes the opposite is required. At times, the correct operation would be to only process the actions of an event policy if another event has not occurred within a set time period. The configuration syntax in both cases is similar, except the keyword `not` is included to indicate that receipt of the correlating event within the time-frame causes the policy not to be triggered:

```
within <seconds> not events [ <correlating events> ]
```

As an example, assume that it is company policy to only perform commits during a two hour maintenance window in the middle of the night. Commits outside of this window should result in syslog error messages that can be followed up on later by operations management. To accomplish this policy a time generate-event is created called `MAINTENANCE-START`. Time generate-events are covered later in this chapter, for now just be aware that every day at 23:00 the `MAINTENANCE-START` event is triggered.

An event policy is created that responds to the UI_COMMIT event. But the policy has a correlating not event of MAINTENANCE-START with a within time of 2 hours. This means that any commit executes the event script, unless the commit is within two hours of the MAINTENANCE-START event. When executed, the event script writes a message to the syslog indicating that a commit was made outside of the approved hours.

Here is the event policy needed to automate this action:

```
event-options {
   policy off-time-commit {
      events ui_commit;
      within 7200 {
         not events maintenance-start;
      }
      then {
         event-script log-syslog-error.slax;
      }
   }
   event-script {
      file log-syslog-error.slax;
   }
}

Here is the log-syslog-error.slax event script that is executed as a result
of this event policy:
/* log-syslog-error.slax */
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 / {
   <event-script-results> {

      /* Send syslog error message */
      expr jcs:syslog("external.error", "Commit performed out of
maintenance window.");
   }
}
```

Matching Event Attributes

A more precise event policy can be crafted by comparing the event attributes against desired values. Most events have one or more attributes that can be referenced within the event policy. When the event policy includes attributes, Junos considers not only the occurrence of a system event, but also the value of the event's attributes as well.

To match event attributes, use the attributes-match statement under the event policy:

```
attributes-match {
   <event1.attribute> equals <event2.attribute>;
   <event1.attribute> starts-with <event2.attribute>;
   <event.attribute> matches <regular-expression>;
}
```

Attribute values can be compared in three ways:

- equals: The two attributes must share the same value.
- starts-with: The first attribute must start with the value of the second attribute.
- matches: The attribute must match the regular expression.

The events referenced in `attributes-match` statements must be either the trigger event or the correlating events included in the event policy's `within` statements. This requirement results in two conventions:

- The `equals` and `starts-with` attribute comparisons can only be used when a `within` statement is present in the event policy.
- The `matches` comparison can be used without a `within` statement, but only if the event referenced is a trigger event for the event policy.

Let's return to the `save-rollback.slax` example to illustrate the usefulness of comparing event attributes. In that example, the following event policy was used:

```
event-options {
   policy faulty-rollback {
      events ui_commit_not_confirmed;
      within 60 events ping_test_failed;
      within 50 events ui_commit_not_confirmed;
      then {
         event-script save-rollback.slax;
      }
   }
   event-script {
      file save-rollback.slax;
   }
}
```

The event policy has a potential problem: the policy only functions correctly when a single RPM test is configured. If multiple RPM tests are running on the Junos device then the event policy, as currently configured, has no way to detect which test has failed.

But, the `PING_TEST_FAILED` event has two attributes that can be used to solve this problem:

- `test-owner` – The RPM probe owner
- `test-name` – The RPM test name

If the event policy is configured to only react when a `PING_TEST_FAILED` event has a specific `test-owner` or `test-name`, then the failure of other RPM tests does not cause the unwanted trigger of the event policy.

Assume, for example, that the RPM configuration uses a probe name of `"Connectivity"` and a test name of `"Management."` The following event policy configuration could be used on the Junos device to prevent conflicts with other RPM tests:

```
event-options {
   policy faulty-rollback {
      events ui_commit_not_confirmed;
      within 60 events ping_test_failed;
      within 50 events ui_commit_not_confirmed;
      attributes-match {
         ping_test_failed.test-owner matches "^Connectivity$";
         ping_test_failed.test-name matches "^Management$";
      }
      then {
```

```
            event-script save-rollback.slax;
        }
    }
    event-script {
        file save-rollback.slax;
    }
}
```

If an event is not mentioned in an `attributes-match` statement then its attributes are not compared. In the above example there are two events referenced in the policy:

- `UI_COMMIT_NOT_CONFIRMED`
- `PING_TEST_FAILED`

When deciding if a `PING_TEST_FAILED` event has occurred within the past 60 seconds (to meet the demands of the `within 60` statement), the two `attributes-match` statements compare the values of the `PING_TEST_FAILED` event's attributes. But, when considering if a `UI_COMMIT_NOT_CONFIRMED` event can act as the trigger event or the received event for the `within 50` statement, no attribute comparisons are performed because that event is not referenced in an `attributes-match` statement.

Matching Nonstandard Events

Attribute matches are useful for many event policies, but they are absolutely essential when trying to react to a nonstandard event. As discussed earlier in this chapter, a nonstandard event does not have a unique event ID. Instead these events share an ID with other nonstandard events of the same origin. The only way for an event policy to differentiate between these events is by comparing their `message` attribute against the desired syslog message.

For example, consider the following syslog message that occurs because of a failed power supply:

```
Jul 29 11:41:13  Junos craftd[1168]:  Major alarm set, Power Supply B not providing power
```

This is a nonstandard event that lacks a specific event ID. The event is originated by the `craftd` daemon, which uses the `SYSTEM` event ID for nonstandard events. An event policy could match on this nonstandard event with the following configuration:

```
event-options {
    policy catch-power-supply-alarm {
        events SYSTEM;
        attributes-match {
            system.message matches "Major alarm set, Power Supply . not providing power";
        }
        then {
            ...
        }
    }
}
```

Try It Yourself: Matching Nonstandard Events

Find a nonstandard event that has been logged to the syslog of your Junos device. Craft an event policy that matches this event and executes an event script. The event script should write a message to the syslog indicating that the script was executed.

Count-Based Triggers

Some event policies should only run when the trigger event(s) have occurred a certain number of times within a configured timeframe. For example: only perform the policy actions if the trigger event(s) have occurred 5 times, or if the trigger event(s) have occurred less than 3 times. This behavior is configured by using the `trigger` statement. This statement requires that a count be specified along with one of the following instructions:

- `on` – Run the policy if the occurrence count equals the configured count.
- `until` – Run the policy if the occurrence count is less than the configured count.
- `after` – Run the policy if the occurrence count exceeds the configured count.

The `trigger` statement is enclosed inside a `within` statement that contains the timeframe:

```
within <seconds> {
   trigger (on # | until # | after # );
}
```

For example, the following event policy is run if the user "jnpr" logs in more than three times within a ten second period:

```
policy lots-of-logins {
   events ui_login_event;
   within 10 {
      trigger after 3;
   }
   attributes-match {
      ui_login_event.username matches jnpr;
   }
   then {
      ...
   }
}
```

Generating Time-Based Events

Rather than wait for a system event to occur, event policies can be run on a time-interval or at a specific time of day. Time-based events are created through the `generate-event` configuration statement. These time-based events exist solely to trigger event policies; they do not result in a syslog message when they are generated.

There are two types of generated events:

- `time-interval` – Event is generated at a specific interval configured in seconds. (Minimum: 60, Maximum: 604800).
- `time-of-day` – Event is generated at a specific time of day configured in 24-hour format: HH:MM:SS.

NOTE Time-of-day events are relative to the local Junos device time.

Here is an example event that is generated every five hours:

```
event-options {
   generate-event {
      every-5-hours time-interval 18000;
   }
}
```

This event is generated at 5:00 am every day:

```
event-options {
   generate-event {
      daily-05:00 time-of-day "05:00:00 +0000";
   }
}
```

NOTE Junos automatically generates the time-zone offset number for the `time-of-day` statement if the offset is not specified. This offset is based on the local time-zone of the Junos device.

Event policies can match these generated events in the same way they match normal events. The policy statement uses the generated event name in place of an event ID:

```
policy match-5-hours {
   events every-5-hours;
   then {
      ...
   }
}
```

ALERT! Junos devices can have a maximum of 10 configured generated events. This includes both `time-interval` as well as `time-of-day` events.

Time-based events allow a Junos device to perform specific actions at certain times. As an example, the below event script logs a syslog warning message if the `/var filesystem` has exceeded 75% utilization. The event script is executed every hour by its event policy.

Here is the configuration for the `generate-event` and event policy:

```
event-options {
   generate-event {
      every-hour time-interval 3600;
   }
   policy check-var-utilization {
      events every-hour;
      then {
         event-script check-var-utilization.slax;
      }
   }
   event-script {
      file check-var-utilization.slax;
   }
}
```

Here is the `check-var-utilization.slax` event script:

```
/* check-var-utilization.slax */
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 / {
   <event-script-results> {

      /* Get show system storage */
      var $system-storage = jcs:invoke( "get-system-storage" );

      /* Retrieve the /var percent */
      var $percent = $system-storage/filesystem[mounted-on=="/var"]/
used-percent;

      /* Is it too high? Then log a warning. */
      if( $percent > 75 ) {
         var $percent-string = normalize-space( $percent ) _ "%";
         var $syslog-message = "Warning: /var utilization is at " _
$percent-string;
         expr jcs:syslog( "external.warning", $syslog-message );
      }
   }
}
```

With this configuration Junos generates the `every-hour` event every hour, which causes the `check-var-utilization` event policy to execute the `check-var-utilization.slax` event script. The script then retrieves the system storage information and logs a warning if the `/var` partition has become highly utilized.

NOTE This script might have to be modified to run correctly on some Junos devices, as not all devices have a `/var mount` point, and "show system storage" returns slightly different results on certain platforms such as the EX4200 Virtual Chassis.

How to Change the Junos Configuration at a Specific Time of Day

The following shows how to configure a Junos device to prefer its ISP-1 BGP provider from 6:00 am to 6:00 pm, and ISP-2 from 6:00 pm to 6:00 am.

1. Configure two separate time-based events: one generated at 6:00 am and one generated at 6:00 pm.

```
generate-event {
   06:00 time-of-day "06:00:00 +0000";
   18:00 time-of-day "18:00:00 +0000";
}
```

2. Configure an event policy to execute `day-policy.slax` to make the needed morning changes to prefer ISP-1.

```
policy prefer-isp-1 {
   events 06:00;
   then {
      event-script day-policy.slax;
   }
}
```

3. Configure an event policy to execute `night-policy.slax` to make the needed evening changes to prefer ISP-2.

```
policy prefer-isp-2 {
   events 18:00;
   then {
      event-script night-policy.slax;
   }
}
```

NOTE Configuration changes are performed by event scripts in the same manner as op scripts. For details on how to perform configuration changes refer to P*art One: Applying Junos Operations Automation*.

Here is the full configuration for the generate-events and event policy:

```
event-options {
   generate-event {
      06:00 time-of-day "06:00:00 +0000";
      18:00 time-of-day "18:00:00 +0000";
   }
   policy prefer-isp-1 {
      events 06:00;
      then {
         event-script day-policy.slax;
      }
   }
   policy prefer-isp-2 {
      events 18:00;
      then {
         event-script night-policy.slax;
      }
   }
   event-script {
      file day-policy.slax;
      file night-policy.slax;
   }
}
```

```
Here are the day-policy.slax and night-policy.slax event scripts:
/* day-policy.slax */
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 / {
   <event-script-results> {

      /* open connection */
      var $connection = jcs:open();

      /* configuration change to prefer ISP-1 */
      var $change = {
         <configuration> {
            <policy-options> {
               <policy-statement> {
                  <name> "isp-1-import";
                  <then> {
                     <local-preference> {
                        <local-preference> 150;
                     }
                  }
               }
               <policy-statement> {
                  <name> "isp-2-import";
                  <then> {
```

```
                        <local-preference> {
                            <local-preference> 50;
                        }
                    }
                }
            }
        }

        /* load and commit the change */
        var $results = {
            call jcs:load-configuration( $connection, $configuration = $change
);
        }

        /* Report any errors or success */
        if( $results//xnm:error ) {
            expr jcs:syslog( "external.error", "Couldn't make policy
changes.");
        }
        else {
            expr jcs:syslog( "external.info", "Import policies now prefer
ISP-1" );
        }

        /* Close the connection */
        expr jcs:close( $connection );
    }
}

/* night-policy.slax */
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 / {
    <event-script-results> {

        /* open connection */
        var $connection = jcs:open();

        /* configuration change to prefer ISP-2 */
        var $change = {
            <configuration> {
                <policy-options> {
                    <policy-statement> {
                        <name> "isp-2-import";
                        <then> {
                            <local-preference> {
                                <local-preference> 150;
                            }
                        }
                    }
                    <policy-statement> {
                        <name> "isp-1-import";
                        <then> {
                            <local-preference> {
                                <local-preference> 50;
                            }
```

```
                    }
                }
            }
        }
    }

    /* load and commit the change */
    var $results = {
        call jcs:load-configuration( $connection, $configuration = $change
);
    }

    /* Report any errors or success */
    if( $results//xnm:error ) {
        expr jcs:syslog( "external.error", "Couldn't make policy
changes.");
    }
    else {
        expr jcs:syslog( "external.info", "Import policies now prefer
ISP-2" );
    }

    /* Close the connection */
    expr jcs:close( $connection );
  }
}
```

At 6:00 am Junos generates the 06:00 event that triggers the `prefer-isp-1` event policy and executes the `day-policy.slax` event script. This script changes the configuration to prefer routes from ISP-1. Then, at 6:00 pm the 18:00 event is generated, triggering the `prefer-isp-2` policy that executes the `night-policy.slax` event script. This event script performs the opposite of the `day-policy.slax` script. Rather than preferring ISP-1, the `night-policy.slax` event script configures Junos to prefer ISP-2 over ISP-1.

NOTE These event scripts assume that the correct routing policy is present when the event policies and event scripts are first added to the Junos device. The event scripts do not check the configuration until one of the time-based events occurs.

Changing the System Time

If the time is manually changed on the Junos device through the `set date` command then all generated events must be recalibrated or they cannot run correctly. To refresh the generated events use one of the following commands:

- `commit full`
- `restart event-processing`
- `request system scripts event-scripts reload`

Try It Yourself: Time-based Configuration Changes

Create a local user account called *test-user* on your Junos device. Create the necessary generated events, event policies, and event scripts to have the test-user automatically assigned to the super-user class from 8am to 5pm and the read-only class from 5pm to 8am.

Chapter 7

Additional Policy Actions

Chapter 6 covered the syntax rules of the SLAX scripting. Chapter 5 demonstrated the use of event policies to automatically execute event scripts in response to system events. This chapter discusses the other actions that an event policy can perform: executing operational commands, uploading files to remote destinations, raising SNMP traps, and ignoring system events.

TIP Multiple actions can be configured within an event policy unless the `ignore` action is being used. When used, `ignore` must be the only action performed by an event policy.

Executing Commands

Operational commands can automatically execute in response to a system event and their outputs can be stored either locally or remotely. This functionality allows Junos to immediately gather relevant information in response to a system event rather than waiting for later troubleshooting by human operators.

To configure this policy action, use the `execute-commands` statement. One or more commands can be specified along with the archive destination to store the outputs.

The following example demonstrates how to use the `execute-commands` statement within an event policy. In this example, the event policy gathers the outputs of a set of OSPF commands anytime an OSPF neighbor goes down:

```
event-options {
    policy ospf-neighbor-down {
        events rpd_ospf_nbrdown;
        then {
            execute-commands {
                commands {
                    "show ospf neighbor extensive";
                    "show ospf interface extensive";
                    "show ospf statistics";
                }
                output-filename ospf-down-output;
                destination local;
                output-format text;
            }
        }
    }
    destinations {
        local {
            archive-sites {
                /var/tmp;
            }
        }
    }
}
```

A number of configuration statements are used with the `execute-commands` statement to control which commands to execute, the output format to use, and the archive destination to store the output. The following sections explain these commands.

Selecting Commands

The commands executed by the event policy are all included under `execute-commands` by using the `commands` statement. One or more commands can be run, but they must

all be operational mode commands.

NOTE Configuration changes have to be made through scripts. Execute-commands can only be used for operational mode commands.

```
event-options {
   policy ospf-neighbor-down {
      events rpd_ospf_nbrdown;
      then {
         execute-commands {
            commands {
               "show ospf neighbor extensive";
               "show ospf interface extensive";
               "show ospf statistics";
            }
            ...
         }
      }
   }
}
```

In the above example, `"show ospf neighbor extensive"`, `"show ospf interface extensive"`, and `"show ospf statistics"` are all executed sequentially.

Configuring a File Destination

When executing commands, the event policy must include configuration statements setting a local or remote destination to store the output of the executed commands. Here is an example of a local destination named local:

```
event-options {
   destinations {
      local {
         archive-sites {
            /var/tmp;
         }
      }
   }
}
```

The destination must also be configured within the `execute-commands` statement:

```
event-options {
   policy ospf-neighbor-down {
      events rpd_ospf_nbrdown;
      then {
         execute-commands {
            ...
            destination local;
            ...
         }
      }
   }
}
```

The `ospf-neighbor-down` event policy example shows how to use local file directories as the output file destination. Alternatively, remote archive locations can be configured by specifying the archive-site as a URL, rather than a local directory.

Valid transmission protocols used to communicate between the Junos device and the remote archive include HTTP, FTP, and SCP. Here is an example of a remote destination named `remote-scp` that uses SCP as the transmission protocol:

```
event-options {
   destinations {
      remote-scp {
         archive-sites {
            "scp://user@10.0.0.1:/var/tmp" password "$9$oSaDk.mT3/
tfTIESyKvaZG";
         }
      }
   }
}
```

MORE? For more information on configuring archive destinations see the *Configuration and Diagnostic Automation Guide* within the Junos documentation at www.juniper.net/techpubs/.

Specifying an Output Filename

An output filename must also be configured within `execute-commands` to indicate the string to use for the output files generated by the policy. This is accomplished by the `output-filename` statement:

```
event-options {
   policy ospf-neighbor-down {
      events rpd_ospf_nbrdown;
      then {
         execute-commands {
            ...
            output-filename ospf-down-output;
            ...
         }
      }
   }
}
```

The actual filename created by the policy is a combination of the following:

- hostname
- `output-filename` string
- date and time
- unique sequence number (if needed to distinguish between files)

These are combined in the following format:

```
hostname_filename_YYYYMMDD_HHMMSS_index-number
```

For example, with a hostname of Junos and an output file name of `ospf-down-output` the event policy could create the following output file names:

```
Junos_ospf-down-output_20090718_111648
Junos_ospf-down-output_20090718_111912
Junos_ospf-down-output_20090718_122021
```

Choosing an Output Format

By default, Junos stores the output of executed commands as XML data. If text output is desired instead, then use the `output-format` statement to select `text` output rather than `xml`:

```
event-options {
   policy ospf-neighbor-down {
      events rpd_ospf_nbrdown;
      then {
         execute-commands {
            ...
            output-format text;
            ...
         }
      }
   }
}
```

Executing as a Different User

The root user executes the commands by default. If the event policy should execute the commands using a different user account then indicate this by including the `user-name` statement:

```
event-options {
   policy ospf-neighbor-down {
      events rpd_ospf_nbrdown;
      then {
         execute-commands {
            ...
            user-name operator;
            ...
         }
      }
   }
}
```

Using Event Policy Variables

When specifying the commands to execute, it is possible to use event policy variables to include event attributes within the commands themselves. Three different variables are supported:

- `{$$.attribute}`: This variable refers to an attribute of the trigger event.
- `{$event.attribute}`: This variable refers to an attribute of the most recent occurrence of the specified event ID.
- `{$*.attribute}`: This variable refers to an attribute of the most recent correlating event from a `within` clause of the event policy.

By including event policy variables, the executed commands can be crafted to gather the specific information desired - based on the actual event that occurred. For example, the following configuration uses an event policy variable to gather the `show interface extensive` output from an interface that just went down:

```
event-options {
   policy save-interface-output {
      events snmp_trap_link_down;
```

```
        then {
            execute-commands {
                commands {
                    "show interfaces extensive {$$.interface-name}";
                }
                ...
            }
        }
    }
    ...
}
```

The {$$.interface-name} event policy variable matches the SNMP_TRAP_LINK_DOWN trigger event, which has an interface-name attribute. For example:

- If so-2/0/0 went down the policy triggers the output of the show interfaces extensive so-2/0/0 command.
- If ge-1/0/1 went down the policy triggers the output of the show interfaces extensive ge-1/0/1 command.

To retrieve attributes from the most recent occurrence of the indicated event, use the {$event.attribute} variable. As an example, assume that a Junos device has a configured time-of-day generate-event called MAINTENANCE-STOP. When that event occurs, if any interface has gone up or down in the past hour–as indicated by a SNMP_TRAP_LINK_UP or SNMP_TRAP_LINK_DOWN event–athen the policy executes the show interfaces command to capture the last interface to go up, as well as the last interface to go down. The below event policy shows a possible configuration:

```
event-options {
    policy save-interface-output {
        events maintenance-stop;
        within 3600 events [ snmp_trap_link_down snmp_trap_link_up ];
        then {
            execute-commands {
                commands {
                    "show interfaces {$snmp_trap_link_up.interface-name}";
                    "show interfaces {$snmp_trap_link_down.interface-name}";
                }
                ...
            }
        }
    }
    ...
}
```

NOTE Versions of Junos prior to 9.6 require the event ID to be capitalized within the event policy variable. Starting with Junos 9.6 the event ID can be configured in either upper or lower case. The attribute name must always be configured in lower case.

TIP The {$event.attribute} variable always selects the most recent occurrence of the event ID. When using an attributes-match statement in the event policy, be aware that the event that fulfilled the attributes-match might not be the most recent occurrence of the event ID.

But what if only the most recent interface to go up or down is needed for the interface output? In that case, the {$*.attribute} variable is the appropriate variable to use. This variable represents an attribute of the most recent correlating event, which could be either SNMP_TRAP_LINK_UP or SNMP_TRAP_LINK_DOWN in the configuration above.

Here is an example of the modification necessary to cause only one interface output, either the interface that went up or the interface that went down, to be included in the event policy output:

```
event-options {
   policy save-interface-output {
      events maintenance-stop;
      within 3600 events [ snmp_trap_link_down snmp_trap_link_up ];
      then {
         execute-commands {
            commands {
               "show interfaces extensive {$*.interface-name}";
            }
            ...
         }
      }
   }
   ...
}
```

TIP The `{$*.attribute}` variable always selects the most recent occurrence of a correlating event. When using an `attributes-match` statement in the event policy, be aware that the event that fulfilled the `attributes-match` might not be the most recent correlating event.

BEST PRACTICE Do not use an event policy variable to refer to a correlating event that is subject to an `attributes-match` for the event policy.

Try It Yourself: Executing Commands

Create an event policy that reacts to a `UI_COMMIT` event by storing the configuration of the user account that performed the commit. The output file should be saved locally in the `/var/tmp` directory in XML format.

Uploading Files

At times, it is appropriate for an event policy to upload an existing file from the Junos device to a remote server. This file could be a configuration, log file, core-dump, or any other local file that should be stored remotely.

To configure this event policy action, use the `upload filename` statement. When doing so, specify the file to copy as well as the destination to which Junos should upload the file. Here is an example of an event policy that uploads any eventd core files to a remote destination in response to an eventd core-dump:

```
event-options {
   policy save-core {
      events kernel;
      attributes-match {
         kernel.message matches "(eventd).*(core dumped)";
      }
      then {
         upload filename /var/tmp/eventd.core* destination remote-host;
      }
   }
   ...
}
```

TIP The remote-host archive destination must also be configured in the same manner as shown in the previous section.

MORE? For more information on the upload file event policy action see the *Configuration and Diagnostic Automation Guide* within the Junos documentation at www.juniper.net/techpubs/.

Try It Yourself: Uploading Files

Create an event policy that uploads the `messages` log file to a remote server every day.

Raising SNMP Traps

In addition to the actions discussed in prior sections, Junos devices can be configured to generate a SNMP trap in response to a system event. The configuration statement to perform this action is `raise-trap`. The format of the SNMP trap is specified in the `jnx-syslog.mib`. The trap contains the syslog message, event ID, attributes and their values, as well as the syslog severity and facility.

Here is a simple example of an event policy that generates a trap every time a commit is performed:

```
event-options {
   policy commit-trap {
      events ui_commit;
      then {
         raise-trap;
      }
   }
}
```

MORE? For more information on the `raise-trap` event policy action see the *Configuration and Diagnostic Automation Guide* within the Junos documentation at www.juniper.net/techpubs/.

Try It Yourself: Raising SNMP Traps

Create an event policy that raises an SNMP trap every time a user enters or leaves the configuration database.

Ignoring Events

In some cases the script should trigger Junos to process an event only a certain number of times. The `ignore` statement causes the event processing daemon to halt event processing for the trigger event. Junos does not process any event policies that occur later in the configuration, and it logs no messages to the syslog for the event.

ALERT! The `ignore` statement must be the only action configured for the policy.

The `ignore` action is typically used along with the `trigger after` matching condition to dampen events. This approach ensures that an event is only processed a certain number of times within the configured timeframe. If the time for processing an event is exceeded, then the event processing daemon ignores any further occurrences of the event within the given timeframe. One example of where dampening would be

useful is a specific event that repeatedly fills the syslog. A policy that causes Junos to ignore the event after a configured amount of occurrences would help to mitigate the impact on system resources. Another example is configuring Junos to only execute the `ospf-neighbor-down` event policy a maximum of five times within a five minute period:

```
event-options {
    policy ignore-ospf-neighbor-down {
        events rpd_ospf_nbrdown;
        within 300 {
            trigger after 5;
        }
        then {
            ignore;
        }
    }
    policy ospf-neighbor-down {
        events rpd_ospf_nbrdown;
        then {
            execute-commands {
                commands {
                    "show ospf neighbor extensive";
                    "show ospf interface extensive";
                    "show ospf statistics";
                }
                output-filename ospf-down-output;
                destination local;
                output-format text;
            }
        }
    }
    destinations {
        local {
            archive-sites {
                /var/tmp;
            }
        }
    }
}
```

The `ignore-ospf-neighbor` policy only runs if the `RPD_OSPF_NBRDOWN` event occurs more than five times within a five minute period. When that happens, the policy action causes the event to be ignored. This means that the following `ospf-neighbor-down` policy does not run and a syslog message is not logged by the event.

Try It Yourself: Ignoring Events

Add an `ignore-event` policy before the event policy that was created in the Executing Commands section. This `ignore-event` policy should run if a commit is performed more than once per minute.

Chapter 8

Event Script Capabilities

The previous chapters showed examples of simple event scripts. As these past examples demonstrated, event scripts are basically op scripts that are executed automatically in response to a system event. Many successful event scripts can be written in that way, but this chapter introduces some unique event script capabilities. These new capabilities allow event scripts to act in ways that an automatically executed op script cannot, and that provides new possibilities for event scripts.

In addition, this chapter discusses general event script specific topics that relate both to true event scripts, which are enabled under event-options event-script, and to automatically executed op scripts, which are enabled under system scripts op. Topics that are specific to event scripts enabled under event-options event-script have that restriction noted within their section.

NOTE See Chapter 5 for the different storage/enable methods.

Executing Event Scripts

Event scripts are automatically executed by configuring them as an action of an event policy:

```
event-options {
   policy example {
      events ui_commit;
      then {
         event-script example-script.slax;
      }
   }
   event-script {
      file example-script.slax;
   }
}
```

More than one event-script can be executed by a single policy:

```
event-options {
   policy example {
      events ui_commit;
      then {
         event-script example-script.slax;
         event-script example-script-2.slax;
      }
   }
   event-script {
      file example-script.slax;
      file example-script-2.slax;
   }
}
```

Event Script Output

Unlike op scripts, event scripts have no way to output text to the user console. Op scripts are executed manually by users, within their user session. This session has a corresponding console that the op script can write text to. In contrast, event scripts are executed automatically by the event processing daemon. They are not executed within a normal user session, so there is no console available for event scripts to write to.

But, an alternate output method exists for event scripts. Event scripts can be configured to write their output to a file, which is stored locally or remotely, in the same way

that the execute-commands event policy action stores its output. The configuration used is also identical to the execute-commands statement discussed in Chapter 7. The steps to write event script output to a file are:

- A destination is configured – pointing at either a local file directory or a remote URL.
- The destination is referenced under the event-script policy action statement along with an output filename string.

As mentioned in Chapter 7, the actual filename is created by combining the Junos device hostname with the configured filename string, the date and time, and a unique index number, if necessary.

Here is an example of the event policy necessary to run the hello-world.slax op script (shown in *Part One: Applying Junos Operations Automation*) as an event script:

```
event-options {
   policy hello-login {
      events ui_login_event;
      attributes-match {
         ui_login_event.username matches "^jnpr$";
      }
      then {
         event-script hello-world.slax {
            output-filename hello-world-output;
            destination local;
            output-format xml;
         }
      }
   }
   destinations {
      local {
         archive-sites {
            /var/tmp;
         }
      }
   }
   event-script {
      file hello-world.slax;
   }
}
```

In this example, the hello-world.slax script is configured to execute anytime the jnpr user logs into the Junos device. The script uses hello-world-output as its output filename string and stores its output files in /var/tmp. The selected output format is XML rather than text.

With a hostname of Junos, the actual output filenames stored in /var/tmp might look similar to this:

```
Junos_hello-world-output_20090803_224719
```

The script itself is very basic: it outputs Hello World! Here is an example of what the output file contents look like (white space has been added for readability):

```
<event-script-results>
   <output>Hello World!</output>
</event-script-results>
```

Output Format and Executing User

The format used for the event script output file is configured using the `output-format` statement in the same way as the `execute-commands` policy action, but the default format for event scripts is `text` rather than `xml`. The `user-name` statement can be configured for the event script as well, in which case the event daemon executes the script using the specified user account rather than the default root user.

ALERT! The `output-format` and `user-name` statements can only be used with event scripts that are enabled under `system scripts op` (see Chapter 1 for the different event script enabling methods). The statements do not function correctly for event scripts enabled under `event-options event-script`. In the latter case, the `xml` output-format is always used whether the `output-format` statement is included or not, and the scripts must be executed by the root user so the `user-name` statement should not be used.

Event Script Arguments

Event scripts can receive arguments at execution time in a similar manner as op scripts. Within the script itself, the arguments are specified by defining parameters of the same name. For example, to receive an argument named `message` the event script would declare the following global parameter :

```
param $message;
```

With op scripts, administrators provide the arguments to the script by entering them on the command-line, but event script arguments are supplied from the event policy configuration itself. The event script can include an `arguments` statement underneath the `event-script` statement and specify multiple arguments:

```
event-options {
   policy log-root-login {
      events login_root;
      then {
         event-script log-message.slax {
            arguments {
               severity notice;
               facility external;
               message "Login by root";
            }
         }
      }
   }
   event-script {
      file log-message.slax;
   }
}
```

In the example above, three separate arguments are provided to the `log-message.slax` script when it executes. To use the arguments, the `log-message.slax` script must have the following global parameters declared:

```
param $severity;
param $facility;
param $message;
```

Arguments may include event policy variables to pass event specific information to the event script as well. Event scripts can use each of the variables available for the `execute-commands` policy action :

- `{$$.attribute}` - This variable refers to an attribute of the trigger event.
- `{$event.attribute}` - This variable refers to an attribute of the most recent occurrence of the specified event ID.
- `{$*.attribute}` - This variable refers to an attribute of the most recent correlating event from a `within` clause of the event policy.

NOTE Versions of Junos prior to 9.6 require the event ID within the event policy variable to be capitalized. Starting with Junos 9.6, the event ID can be configured in either upper or lower case. The attribute name must always be configured in lower case.

TIP The `{$event.attribute}` and `{$*.attribute}` variables always select the most recent occurrence of their particular events. When using an `attributes-match` statement in the event policy, be aware that the event that fulfilled the `attributes-match` might not be the most recent correlating event.

BEST PRACTICE Do not use an event policy variable to refer to a correlating event that is subject to an `attributes-match` for the event policy.

Here is an example of an event policy that passes event policy variables to an event script through arguments. The goal of this event policy and event script is to administratively disable all newly inserted interfaces that do not already have the configuration applied. This is accomplished by passing the name of the newly active interface to the event script, which first checks if any configuration is present for the interface. If the interface is not configured then the event script adds a `disable` statement under the interface and commits the change.

Here is the event policy to perform this action:

```
event-options {
   policy set-admin-down {
      events chassisd_ifdev_create_notice;
      then {
         event-script set-admin-down.slax {
            arguments {
               interface "{$$.interface-name}";
            }
         }
      }
   }
   event-script {
      file set-admin-down.slax;
   }
}
```

And here is the event script:

```
/* set-admin-down.slax */
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";
```

```
/* Event Script Argument */
param $interface;

match / {
   <event-script-results> {

      /* Retrieve the current configuration */
      var $configuration = jcs:invoke( "get-configuration" );

      /* If the interface is not configured then administratively disable it */
      if( jcs:empty( $configuration/interfaces/interface[name==$interface] ) ) {
         var $change = {
            <configuration> {
               <interfaces> {
                  <interface> {
                     <name> $interface;
                     <disable>;
                  }
               }
            }
         }

         /* Load and commit the configuration change */
         var $connection = jcs:open();
         var $results := {
            call jcs:load-configuration( $connection, $configuration = $change );
         }
         var $close-results = jcs:close( $connection );

         /* Report either errors or success */
         if( $results//xnm:error ) {
            for-each( $results//xnm:error ) {
               expr jcs:syslog( "external.error", "Commit Error: ", message );
            }
         }
         else {
            expr jcs:syslog( "external.info", $interface, ".0 has been disabled." );
         }
      }
      else {
         expr jcs:syslog( "external.info", $interface, ".0 is already configured." );
      }
   }
}
```

ALERT! Prior to Junos 9.5R2, in some cases a space would be appended to the argument's value. This occurs to event scripts that are enabled under `event-options event-script` (See Chapter 1 for the different event script enabling methods). Event scripts that receive arguments and are expected to be run on Junos versions prior to 9.5R2 should be written to handle the possible extra space character.

Try it Yourself: Logging Out Users

Using the `clear bgp neighbor` command without specifying a peer address causes all BGP peers to be reset. Write an event policy and event script that automatically disconnects any user who runs this command without including a peer address.

jcs:dampen()

Chapter 7 discussed how to use the `ignore` policy action to dampen event policies. An alternative method that events scripts can use internally to dampen operations is the `jcs:dampen()` function. This function is called with an identifying string tag. The function tracks how often it has seen that tag within a given timeframe and signals to the calling event script if dampening should be applied.

Here is the syntax of `jcs:dampen()`:

```
var $result = jcs:dampen( string-tag, maximum, within-minutes);
```

The `string-tag` argument is a string that uniquely identifies the operation the script needs to dampen. The `maximum` and `within-minutes` arguments determine when dampening should take effect. If `jcs:dampen()` is called with a `string-tag` more than the maximum times allowed within the timeframe specified, then the function returns a boolean value of false. Otherwise, `jcs:dampen()` returns a boolean value of true to signal that dampening should not take effect yet.

As an example, let's redo the `ospf-neighbor-down` event policy that was created at the beginning of Chapter 7. The same outputs are gathered, but now they are gathered through an event script rather than through the `execute-commands` statement. The `jcs:dampen()` function is used within the event script to prevent the outputs from being gathered more than three times within a minute.

Here is the event policy used:

```
event-options {
   policy ospf-neighbor-down {
      events rpd_ospf_nbrdown;
      then {
         event-script gather-ospf-outputs.slax {
            output-filename ospf-output;
            destination local;
            output-format xml;
         }
      }
   }
   event-script {
      file gather-ospf-outputs.slax;
   }
}
```

And here is the event script:

```
/* gather-ospf-outputs.slax */
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 / {
   <event-script-results> {

      /* If dampening is not in effect then gather the outputs */
      if( jcs:dampen( "gather-ospf", 3, 1 ) ) {
         var $neighbor-extensive-rpc = {
            <get-ospf-neighbor-information> {
               <extensive>;
```

```
            }
        }
        var $neighbor-extensive = jcs:invoke( $neighbor-extensive-rpc );
        var $interface-extensive-rpc = {
            <get-ospf-interface-information> {
                <extensive>;
            }
        }
        var $interface-extensive = jcs:invoke( $interface-extensive-rpc );
        var $statistics = jcs:invoke( "get-ospf-statistics-information" );

        /* Copy their XML output to the output file */
        copy-of $neighbor-extensive;
        copy-of $interface-extensive;
        copy-of $statistics;
    }
  }
}
```

The `jcs:dampen()` function is called with a `string-tag` of `"gather-ospf"`, a `maximum` of 3 and a `within-minutes` of 1. This means that if the Junos device records more than three calls to `jcs:dampen()` with that string-tag within a minute, then the function returns false, otherwise it returns true.

In summary, a true result from `jcs:dampen()` indicates that the number has not exceeded the limit, a false result indicates that it has exceeded the limit and dampening should be enforced.

Try it Yourself: Dampening Event Reactions

Chapter 7 included a save-core event policy that demonstrated the `upload filename` policy action. Using that event policy as a guideline, create a policy that executes an event script in response to an eventd core dump. The event script should upload all eventd core files to a remote server, but the action should be dampened by the `jcs:dampen()` function to a maximum of 1 time per minute. When this limit is exceeded a syslog message should be logged instead, indicating that the core upload process was dampened.

Embedded Event Policy

Up to this point, all examples within this book have required that the event policy used to execute an event script be included within the Junos configuration. But, starting with Junos 9.0, it is possible to embed the actual event policy within the event script itself. This offers three main advantages:

- Reduced configuration size: Because the event policy is no longer a part of the configuration file, the `event-options` configuration hierarchy is smaller, especially when multiple event scripts are in use.
- Easier deployment: By integrating the event policy within the event script, installation becomes a matter of copying the script to the Junos device and enabling the script under `event-options`. The actual event policy is distributed within the event script and does not have to be configured on each device.
- Consistent event policies: Because the event policy is embedded within the event script, all Junos devices that enable the script share a consistent policy configuration.

NOTE Embedded event policies are only supported in Junos 9.0 and later and can only be used when the script is enabled under `event-options event-script`. See Chapter 5 for the different storage/enable methods.

$event-definition

Event scripts may contain embedded event policies within a global variable called `$event-definition`. The embedded policy is the XML representation (in SLAX abbreviated format) of the actual event policy and other desired event configuration, all under a top-level element of <event-options>. When a commit occurs, Junos reads through all event scripts enabled under `event-options event-script`, retrieves all embedded event policy from any declared `$event-definition` global variables, and commits the embedded event configuration as part of the normal configuration.

For example, let's use the event policy from the ospf-neighbor-down example:

```
event-options {
   policy ospf-neighbor-down {
      events rpd_ospf_nbrdown;
      then {
         event-script gather-ospf-outputs.slax {
            output-filename ospf-output;
            destination local;
            output-format xml;
         }
      }
   }
}
```

The following command shows the XML format of this configuration:

```
user@Junos> show configuration event-options policy ospf-neighbor-down |
display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.1I0/junos">
   <configuration junos:commit-seconds="1250877249" junos:commit-
localtime="2009-08-21 10:54:09 PDT" junos:commit-user="jnpr">
         <event-options>
            <policy>
               <name>ospf-neighbor-down</name>
               <events>rpd_ospf_nbrdown</events>
               <then>
                  <event-script>
                     <name>gather-ospf-outputs.slax</name>
                     <output-filename>ospf-output</output-filename>
                     <destination>
                        <name>local</name>
                     </destination>
                     <output-format>xml</output-format>
                  </event-script>
               </then>
            </policy>
         </event-options>
   </configuration>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>
```

Converting the displayed XML output into the SLAX abbreviated format gives

```
us the following:
<event-options> {
    <policy> {
        <name> "ospf-neighbor-down";
        <events> "rpd_ospf_nbrdown";
        <then> {
            <event-script> {
                <name> "gather-ospf-outputs.slax";
                <output-filename> "ospf-output";
                <destination> {
                    <name> "local";
                }
                <output-format> "xml";
            }
        }
    }
}
```

Here is the `gather-ospf-outputs.slax` script, this time using an embedded event policy:

```
/* gather-ospf-outputs.slax */
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";

var $event-definition = {
    <event-options> {
        <policy> {
            <name> "ospf-neighbor-down";
            <events> "rpd_ospf_nbrdown";
            <then> {
                <event-script> {
                    <name> "gather-ospf-outputs.slax";
                    <output-filename> "ospf-output";
                    <destination> {
                        <name> "local";
                    }
                    <output-format> "xml";
                }
            }
        }
    }
}

match / {
    <event-script-results> {

        /* If dampening is not in effect then gather the outputs */
        if( jcs:dampen( "gather-ospf", 3, 1 ) ) {
            var $neighbor-extensive-rpc = {
                <get-ospf-neighbor-information> {
                    <extensive>;
                }
            }
            var $neighbor-extensive = jcs:invoke( $neighbor-extensive-rpc );
            var $interface-extensive-rpc = {
                <get-ospf-interface-information> {
                    <extensive>;
                }
            }
```

```
            var $interface-extensive = jcs:invoke( $interface-extensive-rpc
);
            var $statistics = jcs:invoke( "get-ospf-statistics-information"
);

            /* Copy their XML output to the output file */
            copy-of $neighbor-extensive;
            copy-of $interface-extensive;
            copy-of $statistics;
        }
    }
}
```

Embedding the event policy within the event script allows the policy to be removed from the configuration, because Junos now reads the needed policy from the script itself at commit time. Other than the local destination, the only configuration needed to run the above event script is to enable the script under `event-options event-script`:

```
event-options {
    event-script {
        file gather-ospf-outputs.slax;
    }
}
```

TIP The event script can embed the destination configuration within the `$event-definition` variable along with the `ospf-neighbor-down` event policy if desired.

In addition to the event policies, an event script can also embed additional event configuration. For example, this rewrite of the `check-var-utilization.slax` script embeds both its event policy and the hourly generated event:

```
/* check-var-utilization.slax */
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";

var $event-definition = {
    <event-options> {
        <generate-event> {
            <name> "every-hour";
            <time-interval> "3600";
        }
        <policy> {
            <name> "check-var-utilization";
            <events> "every-hour";
            <then> {
                <event-script> {
                    <name> "check-var-utilization.slax";
                }
            }
        }
    }
}

match / {
    <event-script-results> {

        /* Get show system storage */
        var $system-storage = jcs:invoke( "get-system-storage" );
```

```
      /* Retrieve the /var percent */
      var $percent = $system-storage/filesystem[mounted-on=="/var"]/
used-percent;

      /* Is it too high? Then log a warning. */
      if( $percent > 75 ) {
         var $percent-string = normalize-space( $percent ) _ "%";
         var $syslog-message = "Warning: /var utilization is at " _
$percent-string;
         expr jcs:syslog( "external.warning", $syslog-message );
      }
   }
}
```

Viewing Embedded Policies

To view the embedded event configuration use the following CLI command:

```
show event-options event-scripts policies
```

This command displays the event configuration embedded within any enabled event scripts on the Junos device:

```
user@HOST> show event-options event-scripts policies
## Last changed: 2009-07-28 14:10:39 UTC
event-options {
   policy ospf-neighbor-down {
      events rpd_ospf_nbrdown;
      then {
         event-script gather-ospf-outputs.slax {
            output-filename ospf-output;
            destination local;
            output-format xml;
         }
      }
   }
}
```

Refreshing Embedded Policies

When it is time to alter an embedded policy, the actual event script must be edited and then copied over the old version on the Junos device. But this action by itself does not reload the event policy. To refresh the embedded event configuration from enabled event scripts, use one of the following commands:

- `request system scripts event-scripts reload`: Operational mode command that refreshes embedded configuration.
- `commit full`: Configuration mode command that recommits the entire configuration including any embedded configuration.
- `restart event-processing`: Operational mode command that restarts the event processing daemon.

For example:

```
user@HOST> request system scripts event-scripts reload
Event scripts loaded
```

Try it Yourself: Embedding Event Policy

Choose an event policy and event script that you created in one of the prior Try it Yourself exercises. Remove the event policy from the configuration and embed the policy within the event script.

<event-script-input>

An earlier section of this chapter explained how event script arguments can pass information from the event attributes into the script. Starting in Junos 9.3, a more comprehensive method is available to retrieve information about the trigger event and received events. This data is now provided to event scripts within an `<event-script-input>` element that is included in the XML source tree provided to event scripts at their execution time.

NOTE Starting in Junos 9.3 the `<event-script-input>` element is available to event scripts, but only for those enabled under `event-options event-script`. See Chapter 1 for the different event script enabling methods.

Source Tree

The source tree is the reverse of the result tree. As explained in *Part One: Applying Junos Operations Automation*, the result tree is a XML data structure built by a script and delivered to Junos for processing at script termination. With a source tree, however, it is Junos that provides the XML data structure to the script, and this is done when the script first starts rather than when it terminates.

No mention was made of source trees in Part One because that section focuses on op scripts and op scripts do not receive any source tree data at startup.

The event script can extract information from this source tree data in the same way as it does with any other XML data structure and the next section demonstrates how to do this.

Trigger Event Data

Here is an example of the `<event-script-input>` element for an event script that was triggered by the `UI_COMMIT` event:

```
<event-script-input>
   <trigger-event>
      <id>UI_COMMIT</id>
      <type>syslog</type>
      <generation-time junos:seconds="1245311597">2009-06-18 07:53:17
UTC</generation-time>
      <process>
         <name>mgd</name>
         <pid>8720</pid>
      </process>
      <hostname>Junos</hostname>
      <message>UI_COMMIT: User 'user' requested 'commit' operation
(comment: none)</message>
      <facility>interact</facility>
      <severity>notice</severity>
      <attribute-list>
         <attribute>
```

```
            <name>username</name>
            <value>user</value>
         </attribute>
         <attribute>
            <name>command</name>
            <value>commit</value>
         </attribute>
         <attribute>
            <name>message</name>
            <value>none</value>
         </attribute>
      </attribute-list>
   </trigger-event>
</event-script-input>
```

The <trigger-event> child element contains information about the event that triggered the event policy, including the following:

- id: The event ID.
- type: The type of event (syslog, pseudo, internal).
- generation-time: When the event occurred.
- process: The name of the process that logged the event (and PID if appropriate).
- hostname: The hostname of the event source.
- message: The syslog message logged for the event.
- facility: The syslog facility of the event.
- severity: The syslog severity of the event.
- attribute-list: If present, this list includes all event attributes .
- attribute name: The name of the attribute.
- attribute value: The value of the attribute.

Location paths should refer to the top-level <event-script-input> element in order to extract information from the source tree. Here are examples of code to retrieve information from the above <event-script-input> example:

Retrieving the process name:

```
var $process-name = event-script-input/trigger-event/process/name;
```

Retrieving the syslog facility:

```
var $facility-name = event-script-input/trigger-event/facility;
```

Retrieving the username that performed the commit:

```
var $committing-user = event-script-input/trigger-event/attribute-list/
attribute[name=="username"]/value;
```

Not all events have attributes. Here is an example of an <event-script-input> for an event policy that matched on a generated event called minute:

```
<event-script-input>
   <trigger-event>
```

```
      <id>MINUTE</id>
      <type>internal</type>
      <generation-time junos:seconds="1248795322">2009-07-28 15:35:22
UTC</generation-time>
      <process>
         <name>eventd</name>
         <pid>492</pid>
      </process>
      <hostname>Junos</hostname>
      <message>Internal event created by eventd</message>
      <facility>user</facility>
      <severity>notice</severity>
      <attribute-list>
      </attribute-list>
   </trigger-event>
</event-script-input>
```

Received Event Data

In addition to the trigger event, each event policy `within` statement must have a matching received event in order for the event policy to be processed.

TIP Each `within` statement results in a single received event. The received event is the most recent of any of the potential correlating events for the `within` statement. If multiple `within` statements are included then multiple received events are present, one per `within` statement.

These received events are also included in the `<event-script-input>` source tree element. Each event is included in a separate `<received-event>` element, which is a child of the `<received-events>` element.

As an example, consider an event policy with the following `events` statement and `within` statements:

```
policy example {
   events rpd_task_begin;
   within 300 events ui_commit;
   within 200 events chassisd_ifdev_detach_fpc;
   ...
}
```

This would result in an `<event-script-input>` element similar to this:

```
<event-script-input>
   <trigger-event>
      <id>RPD_TASK_BEGIN</id>
      <type>syslog</type>
      <generation-time junos:seconds="1245401205">2009-06-19 08:46:45
UTC</generation-time>
      <process>
         <name>rpd</name>
         <pid>41439</pid>
      </process>
      <hostname>Junos</hostname>
      <message>RPD_TASK_BEGIN: Commencing routing updates, version
9.3R2.8, built 2008-12-17 22:48:00 UTC by builder</message>
      <facility>daemon</facility>
      <severity>notice</severity>
      <attribute-list>
      </attribute-list>
```

```
   </trigger-event>
   <received-events>
      <received-event>
         <id>CHASSISD_IFDEV_DETACH_FPC</id>
         <type>syslog</type>
         <generation-time junos:seconds="1245401202">2009-06-19 08:46:42
UTC</generation-time>
         <process>
            <name>chassisd</name>
            <pid>41437</pid>
         </process>
         <hostname>Junos</hostname>
         <message>CHASSISD_IFDEV_DETACH_FPC: ifdev_detach(9)</message>
         <facility>daemon</facility>
         <severity>notice</severity>
         <attribute-list>
            <attribute>
               <name>fpc-slot</name>
               <value>9</value>
            </attribute>
         </attribute-list>
      </received-event>
      <received-event>
         <id>UI_COMMIT</id>
         <type>syslog</type>
         <generation-time junos:seconds="1245401153">2009-06-19 08:45:53
UTC</generation-time>
         <process>
            <name>mgd</name>
            <pid>34740</pid>
         </process>
         <hostname>Junos</hostname>
         <message>UI_COMMIT: User 'user' requested 'commit' operation
(comment: none)</message>
         <facility>interact</facility>
         <severity>notice</severity>
         <attribute-list>
            <attribute>
               <name>username</name>
               <value>user</value>
            </attribute>
            <attribute>
               <name>command</name>
               <value>commit</value>
            </attribute>
            <attribute>
               <name>message</name>
               <value>none</value>
            </attribute>
         </attribute-list>
      </received-event>
   </received-events>
</event-script-input>
```

Received events contain the same information as the trigger event, and the script can retrieve information in the same way.

For example, retrieving the `CHASSISD_IFDEV_DETACH_FPC` slot number:

```
var $slot = event-script-input/received-events/received-event[id=="CHASSISD_
IFDEV_DETACH_FPC"]/attribute-list/attribute[name==fpc-slot]/value;
```

Here is an example of a script that takes advantage of the <event-script-input> data to accomplish its stated goal. The script re-logs syslog messages at a higher severity level. This could be useful in situations where a network management system should receive all syslog messages of a moderate severity along with one or two messages logged at a low severity level. For example, by default Junos logs the SNMP_TRAP_LINK_UP event at the info level, whereas it logs the SNMP_TRAP_LINK_DOWN event at the notice level. If a syslog server only receives notice level messages then it does not receive the indication that the link came back up.

This event script provides a potential solution to this problem. Using the two events mentioned above as an example, the script can re-log the SNMP_TRAP_LINK_UP event using notice severity rather than its standard info severity level.

NOTE This script only reproduces the syslog message text. The full structured format of the original syslog message is not replicated.

The only event configuration required is enabling the event script:

```
event-options {
   event-script {
      file change-syslog-severity.slax;
   }
}
```

Here is the event script, with its embedded event policy:

```
/* change-syslog-severity.slax */
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";

/* Embedded event policy */
var $event-definition = {
   <event-options> {
      <policy> {
         <name> "change-syslog-severity";
         <events> "snmp_trap_link_up";
         <then> {
            <event-script> {
               <name> "change-syslog-severity.slax";
            }
         }
      }
   }
}

match / {
   <event-script-input> {

      /* Record the facility */
      var $facility = event-script-input/trigger-event/facility;

      /* Get the process-name */
      var $process-name = event-script-input/trigger-event/process/name;
```

```
        /* Get PID */
        var $pid = event-script-input/trigger-event/process/pid;

        /* Get the syslog message */
        var $message =event-script-input/trigger-event/message;

        /* Assemble message */
        var $final-message = {
            expr $process-name;

            /* If they have a PID then include it */
            if( string-length( $pid ) > 0 ) {
                expr [" _ $pid _ "]";
            }

            expr ": ";
            expr $message;
        }

        /* New priority */
        var $new-priority = $facility _ ".notice";

        /* Now re-syslog it with the new facility */
        expr jcs:syslog( $new-priority, $final-message );
    }
}
```

The event script pulls the `facility`, `process-name`, `process-id`, and `syslog message` from `<event-script-input>`. The script then relogs the same syslog message at the notice severity level rather than the info severity level.

NOTE When using this approach, do not attempt to re-log events generated by the cscript process. This is the process used by scripts to create syslog messages. Re-logging these events would result in a loop, as the re-logged event is then re-logged, etc.

Try it Yourself: Using <event-script-input>

Revise your earlier event script that automatically logged out users using the `clear bgp neighbor` command without specifying a peer address. Remove the event policy from the configuration and embed the policy within the event script's `$event-definition` variable. Remove any command-line arguments used to communicate which user performed the command and instead use the `<event-script-input>` source tree element to determine which user should be logged out.ih

Part Three

Applying Junos Configuration Automation

Chapter 9

Introducing Commit Scripts

The Junos automation toolset is a standard part of the Junos operating system available on all Junos platforms including routers, switches, and security devices. This part continues teaching the core concepts of Junos automation begun in the first two parts: Applying Junos Operations Automation and Applying Junos Event Automation. The first part explains the SLAX scripting language and describes how to use op scripts, one type of Junos automation script. The second part explains how to automate events through event policies and scripts. This third part of the book, describes how commit scripts automate the commit process, giving administrators control over what configuration is applied to their Junos device.

Junos Automation Overview

Junos automation enables an organization to embed its wealth of knowledge and experience of operations directly into Junos devices:

- Business rules automation - enforces best practices and changes management to avert human factors.
- Provisioning automation - simplifies and abstracts complex configurations to minimize errors.
- Operations automation - customizes command output to streamline operation and troubleshooting.
- Event automation - performs automatic changes and responses in reaction to observed events.

Through automation, Junos empowers network operators to scale by simplifying complex tasks, maximizing uptime, and optimizing operational efficiency.

Commit Scripts

Parts One and Two of this book, discussed the application of op and event scripts. An op script is a customized command that administrators can execute from the CLI prompt (and other scripts can call) in the same way as a standard Junos command. Op scripts can gather and display desired information, perform controlled configuration changes, or execute a group of operational commands. Event scripts are used in conjunction with event policies to automate reactions to events.

Commit scripts fill the third role of Junos automation by providing a way to add customized intelligence as part of the commit process. Junos executes the commit scripts automatically each time an administrator commits the configuration. Commit scripts can control the commit process in multiple ways ranging from simple warning messages to complex configuration changes based on the presence of configuration macros.

Commit Script Examples

A few examples of how commit scripts can control configuration include:

- Verifying essential configuration hierarchies whenever a user changes the configuration. If a user accidently deletes any of these hierarchies the script can instruct Junos to halt the commit process and issue an error message. This prevents network outages caused by human error.

- Checking for descriptions of interfaces or BGP peers at each commit time. An improperly formatted description can result in a warning message, advising the user to fix it.
- Reducing the number of statements required in complex configurations. Configuration macros can act as customer-standardized configuration syntax and expand into complex configuration structures at commit time.
- Enforcing scaling limits for critical settings. For example, a script can generate a commit error (or warning) when the configuration exceeds the maximum number of permitted peers.

MORE? To see more script examples go to the online script library at www.juniper.net/scriptlibrary.

Commit Process

The Junos commit process not only enables administrators to preview all changes before performing a commit, it also enables Junos to validate the syntax and logic of the candidate configuration before applying it. When a commit is requested, Junos first performs inheritance on the candidate configuration by integrating any configuration groups into their destination hierarchies and removing all inactive statements. The post-inheritance configuration is then checked out by Junos for any configuration errors or warnings. Both errors and warnings are displayed to the committing administrator, but errors also cause Junos to fail the commit.

Commit scripts give users a way to customize the validation process of their configurations in accordance with their own practices and policies. Junos integrates commit scripts seamlessly into its commit process, allowing the scripts to check and modify the configuration prior to the final verification performed by Junos. Figure 9.1 shows where commit scripts fit in the commit process.

The execution of commit scripts occurs after inheritance has been performed on the configuration, so all commit scripts are provided the post-inheritance configuration at the start of their processing. Each commit script refers to this post-inheritance configuration as it determines what actions, if any, need to be performed. A script can perform (the following) five commit-specific actions:

- Displaying a warning to the committing user (Chapter 10).
- Logging a message to the syslog (Chapter 10).
- Generating a commit error and canceling the commit (Chapter 10).
- Changing the configuration (Chapters 11 and 12).
- Transiently changing the configuration (Chapters 11 and 12).

Commit scripts communicate necessary actions to Junos by including instructions in their result tree. Each commit script is executed sequentially and its result tree is provided to Junos when the script terminates. Once all commit scripts have been executed, Junos then processes all of the scripts' instructions. Based on these instructions, Junos might halt the commit, display warning messages, or alter the configuration.

If the commit process is not halted by a commit script, then Junos applies all the commit script changes and performs its final inspection of the checkout configura-

tion. For a configuration that passes all checks, Junos activates the new configuration to the device.

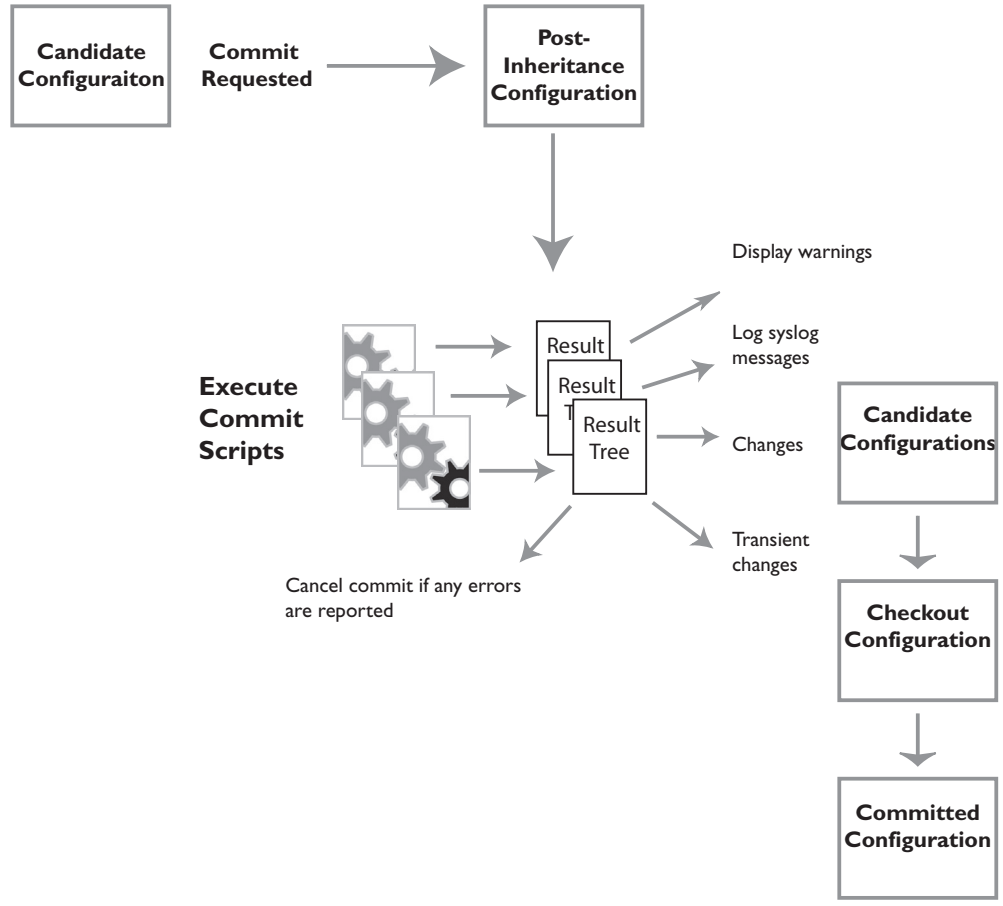


Figure 9.1 Commit Model with Commit Scripts

The placement of commit scripts within the commit process, and their ability to inject instructions to influence that process, provides administrators with flexibility and complete control over the final configurations committed on Junos devices.

Configuration/Storage

Administrators add new commit scripts by enabling them within the configuration and storing them on the Junos device in the /var/db/scripts/commit directory. The command to enable new scripts is:

set system scripts commit file example-commit-script.slax

And the locations are:

- Storage Location: `/var/db/scripts/commit`
- Configuration Location: `[edit system scripts commit]`

NOTE Beginning in Junos 9.4, the Juniper EX Series began storing configurations in /config/db/scripts rather than /var/db/scripts. /var/db/scripts links to the new /config/db/scripts directory so files that are copied into /var/db/scripts are still placed in the correct location.

NOTE The load-scripts-from-flash configuration command changes the script storage location from /var/db/scripts to /config/scripts. If you are using this configuration statement, store the commit scripts in /config/scripts/commit.

Only super-users, users with the all permission bit, or users that have been given the maintenance permission bit are permitted to enable or disable Junos commit scripts in the configuration.

ALERT! The Junos management process executes commit scripts with root permissions, not the permission level of the committing user. If the user has the necessary access permissions to commit the configuration, then Junos performs all actions of the configured commit scripts, regardless of the privileges of the committing user.

NOTE In devices with multiple routing-engines (including EX4200 Virtual Chassis), each routing engine performs the commit process separately. Because of this, the script file must be copied into the commit script directory, and enabled on every routing-engine. Typically the configuration of all routing-engines is done automatically through configuration synchronization, but if the configurations are not synchronized between routing-engines then the script must be enabled on all routing-engines manually.

allow-transients

Commit scripts can perform configuration changes that affect the operation of the Junos device but do not appear in the configuration file. These changes are referred to as transient configuration changes and can be used in conjunction with configuration macros to create custom configuration syntax. Transient configuration changes are discussed in Chapters 11 and 12 and are not permitted by default. To allow them, enter the following configuration command:

```
set system scripts commit allow-transients
```

Commit Script Boilerplate

When writing Junos automation scripts, it is always best to work from the standard boilerplate. This greatly simplifies script writing, as there is no need to memorize the necessary name-space URLs. Instead, just copy and paste the boilerplate and add your script code within it. The boilerplate used for writing commit scripts is similar to the boilerplate used for op and event scripts with two significant differences that are discussed in following sections. Here is the boilerplate that should be used when writing commit scripts:

```
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 {

    /* Your script code goes here */

}
```

- version: While version 1.0 is currently the only available version of the SLAX language, the version line is required at the beginning of all Junos scripts.
- ns: a ns statement defines a namespace prefix and its associated namespace URL. The following three namespaces must be included in all Junos scripts:
  - `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";`

TIP It is easiest to just copy and paste these namespaces into each new script as part of the boilerplate rather than trying to type them out by hand.

import: The import statement is used to import code from one script into the current script. As the `junos.xsl` script contains useful default templates and parameters, all scripts should import this file. The `import "../import/junos.xsl";` line from the boilerplate is all a script needs to accomplish this.

match configuration: This code block is the main template of the commit script.

<commit-script-input>

As discussed in the previous section, there are two significant differences between the boilerplates used for op and event scripts and the boilerplate used for commit scripts. Both of these differences exist to simplify commit scripts and are discussed in this and subsequent sections.

The first difference between the boilerplates is that op and event scripts start with a `match /` main template, while commit scripts begin their processing in a `match configuration` main template.

The match configuration template is used by commit scripts because it simplifies the retrieval of XML data from the post-inheritance candidate configuration provided to commit scripts in their source tree within a top-level element named <commit-script-input>. This simplification allows configuration information to be retrieved by referencing it starting with its top-level configuration hierarchy. For example, the system host-name can be retrieved with the following code:

```
var $configured-host-name = system/host-name;
```

NOTE The source tree was discussed in volume two of this series, Day One: Applying Junos Event Automation. For event scripts, the source tree consists of a top-level element named <event-script-input>. Location paths in event scripts that refer to this element must include the entire path:

```
var $process-name = event-script-input/trigger-event/process/name;
```

Contained within the <commit-script-input> source tree element is a child element named <configuration> that holds the entire post-inheritance candidate configura-

tion. If commit scripts used a `match /` template like op and event scripts do, then it would be necessary to include the full source tree path to the desired configuration data:

```
var $location = commit-script-input/configuration/snmp/location;
```

Commit scripts routinely make extensive queries into the configuration data, and requiring the full path to be specified would be tedious for script writers. The solution was to create a separate `match /` template within the junos.xsl file:

```
<xsl:template match="/">
   <commit-script-results>
      <xsl:apply-templates select="commit-script-input/configuration"/>
   </commit-script-results>
</xsl:template>
```

The above code from the junos.xsl import file is written in XSLT, but here is the corresponding SLAX code:

```
match / {
   <commit-script-results> {
      apply-templates commit-script-input/configuration;
   }
}
```

Because all scripts import the junos.xsl file, this default template becomes part of every commit script's code.

NOTE Op and event scripts observe no ill effect from the inclusion of a match / template in the junos.xsl file because their local match / template overrides the imported template.

When a script first begins, Junos searches the code for a template that matches the current source tree node. The first node checked is the root node, which matches the `match /` template causing Junos to begin executing the code found within that template.

If a specific node is desired then this process can be repeated by using the `apply-templates` statement, which causes the script engine to search for a template that matches the provided location path. In the statement shown above, `apply-templates commit-script-input/configuration` is instructing Junos to find a template that matches the `<configuration>` node, namely, the `match configuration` template that begins commit script operation.

MORE? For more information on unnamed match templates (for example, match configuration) see the *Configuration and Diagnostic Automation* manual of the Junos documentation at www.juniper.net/techpubs/.

Whichever node matches the template's match statement becomes the default context node within that template. All location paths start with a reference point of that default context node, unless they are based on a node-set variable, are based on a function result, or appear within `for-each` loops.

NOTE In the case of event scripts, using a `match /` main template causes the context node to be the root node, so all location paths use the full path to the desired information:

```
var $process-name = event-script-input/trigger-event/process/name;
```

Because commit scripts start with a context node of <configuration>, their location paths can begin with the relevant top-level configuration hierarchy:

```
var $location = snmp/location;
```

This is equivalent to the following root-based path into the Junos candidate configuration:

```
/commit-script-input/configuration/snmp/location
```

Post-inheritance Configuration

As mentioned previously, the candidate configuration provided to commit scripts within the <commit-script-input> source-tree element is the post-inheritance view. This has two implications:

- The configuration groups inherit into the configuration hierarchies, and the [edit groups] hierarchy is not included in the post-inheritance configuration.
- No inactive statements are present in the post-inheritance configuration.

Typically this is the behavior a commit script expects because it is only concerned with the actual configuration that will be committed onto the device. But some commit scripts need to verify the [edit groups] hierarchy and/or view the inactive statements present in the configuration. In those scenarios, the non-inherited candidate configuration must be requested through the <get-configuration> API element:

```
var $configuration = jcs:invoke( "get-configuration" );
```

The resulting variable can then be parsed to pull out whatever configuration data is required:

```
var $re0-group = $configuration/groups[name == "re0"];
```

ALERT! Do not use the <get-configuration> API element within a commit script prior to the following releases: 9.3S5, 9.4R4, 9.5R3, 9.6R2, or 10.0R1, as it can cause the Junos device to hang while booting. For further details refer to PR 452398.

junos:changed

The post-inheritance configuration provided to commit scripts indicates what changes are present in the candidate configuration by including the junos:changed attribute, with a value of "changed", on all changed nodes as well as on their parent and ancestor nodes.

```
<system junos:changed="changed"> {
   <host-name junos:changed="changed"> "juniper1";
   <login> {
      <user> {
         <name> "admin";
         <uid> "2001";
         <class> "superuser";
         <authentication> {
            <encrypted-password> "$1$usVGUcj1$JA/9xEqNrFDImo02nP9tN.";
         }
      }
   }
```

```
    <services> {
        <ssh>;
    }
    <syslog> {
        <file> {
            <name> "messages";
            <contents> {
                <name> "any";
                <any>;
            }
        }
    }
}
```

The above example shows the post-inheritance configuration of a commit script in which the device's hostname has been changed. The `junos:changed` attribute on both the `<host-name>` element as well as on its parent `<system>` element indicates this change.

ALERT! When a Junos daemon generates a commit warning, it does not remove the `junos:changed` attributes from the changed elements, and they persist through following commits. This makes the attribute unreliable as an indication of new changes because it actually reflects changes that occurred since the last daemon-warning-free commit, rather than changes that occurred since the last commit. (Commit script `<xnm:warning>` messages, discussed in Chapter 10, do not cause the same impact to the `junos:changed` attribute that Junos daemon warning messages do).

junos:group

Configuration groups are used as repositories of common configuration statements that inherit into multiple configuration hierarchies. Although the group hierarchy is not present in the post-inheritance configuration provided to commit scripts, it is possible to determine the effect that groups had on the configuration by looking for the `junos:group` attribute. This attribute is included on every configuration element that originated from a configuration group. The value of the `junos:group` attribute is the configuration group name it was inherited from.

```
<snmp junos:group="re0"> {
    <community junos:group="re0"> {
        <name junos:group="re0"> "public";
        <authorization junos:group="re0"> "read-only";
    }
}
```

In this example, the entire `[edit snmp]` hierarchy has been inherited from the re0 configuration group, as reflected by the presence of the `junos:group` attribute in every configuration element with a value of `"re0"`.

<commit-script-results>

In addition to using a different match template, the commit script boilerplate also differs from op and event scripts because it does not include a result tree top-level element. Op scripts have `<op-script-results>`, and event scripts have `<event-script-results>`, so why doesn't the commit script boilerplate include a `<commit-`

script-results> element?

The answer is that this results element is not necessary within the boilerplate because the match / template in the junos.xsl import file automatically writes it to the result tree. The template was shown earlier in this chapter:

```
match / {
    <commit-script-results> {
        apply-templates commit-script-input/configuration;
    }
}
```

Commit script execution begins with this template, causing the <commit-script-results> element to always be written as the top-level element of the result tree. The match configuration boilerplate template is called through the apply-templates statement, and all elements written to the result tree by the commit script are enclosed correctly within <commit-script-results>.

As an example, two common result tree elements used by commit scripts are <xnm:warning> and <xnm:error> (both are discussed in Chapter 10), and a commit script that has the following match configuration template:

```
match configuration {
    <xnm:warning> {
        <message> "This is a warning message.";
    }
    <xnm:error> {
        <message> "This is an error message.";
    }
}
Creates this result tree:
<commit-script-results> {
    <xnm:warning> {
        <message> "This is a warning message.";
    }
    <xnm:error> {
        <message> "This is an error message.";
    }
}
```

Result Tree Interaction

All script types can create a result tree, but commit scripts make more extensive use of this communication path to Junos than either op or event scripts. Op and event scripts only use the result tree to display output. Most communication between Junos and op and event scripts occurs through functions, not through the result tree.

However, the inclusion of commit scripts within the commit process makes their result tree output extremely important. The result tree provides the instructions for Junos to change the configuration, to display warning messages, to log syslog messages, or to cancel the commit with an error. While commit scripts can use the same functions and can interact with the Junos API in the same manner as op and event scripts, this use is not common. The typical commit script performs its processing by using only the post-inheritance candidate configuration as input and writing its instructions for Junos to the result tree.

Boot-up Commit

As a Junos devices boots, it must perform a commit to initialize its daemons with the stored configuration. As this is a special commit, you should consider how any new commit scripts work during this process.

This boot-up commit follows the same pattern as the standard commit process, including the execution of configured commit scripts. There are two important implications of the boot-up commit:

- Some information, such as chassis components, is not available during the boot-up commit.
- Commit errors during the boot-up commit cause the device to boot with no configuration.

To illustrate the above points, consider chassis components. When the configuration is first committed during the boot-up process the chassis components have not yet been initialized, so any chassis-related information is not available. If a commit script generates a commit error due to this lack of information, then the device boots with no configuration. The device remains in this state until an operator manually resolves the problem.

Keep this in mind as you read through the following chapters and think of ways to use commit scripts within your own network. Part of your commit script design process should include consideration of the boot-up commit and how your commit script responds to it.

Commit Script Checklist

The following questions are recommended as checks to perform when adding a new commit script:

1. Has the script been copied to `/var/db/scripts/commit` on all routing-engines

2. Has the script been enabled under [edit system scripts commit] on all routing-engines?

3. If transient changes are used, is allow-transients configured under [edit system scripts commit]?

4. If Junos API information requests are performed, does the script work correctly during the boot-up commit

5. If Junos API information requests are performed, does the script result in a consistent configuration between the master routing-engine and all other routing-engines? (The other routing-engines might not have access to the same information as the master.)

Chapter 10

Commit Feedback and Control

Including commit scripts within the commit process provides the ability to alter the commit process or to provide feedback to the committing user through notifications and warnings. Junos op and event scripts use the <output> result tree element or the jcs:output() function to display output on the console or within their output file respectively, but commit scripts cannot use these methods to deliver messages to committing users. Instead, commit scripts use one of three result tree elements to provide feedback as well as commit control: <xnm:warning>, <syslog>, and <xnm:error>. This chapter discusses these three result tree elements, their relevant child elements, and how to use them.

<xnm:warning>

As the name implies, the <xnm:warning> element causes a warning message to be displayed to the console of the committing user. The commit process is not affected by <xnm:warning> messages and completes successfully unless other errors are found. Here are some possible uses for <xnm:warning> messages:

- Drawing attention to configuration problems that should be corrected.
- Indicating that the commit script is making an automatic change or performing another action that the user should be aware of.
- Informing the committing user that the commit script is not performing its usual actions due to a problem with the configuration, system, or Junos version in use.

To display a warning message, write the <xnm:warning> element to the result tree with a child element of <message> that contains the text to display. So using the traditional Hello World! example, the following code:

```
match configuration {
    <xnm:warning> {
        <message> "Hello World!";
    }
}
```

... displays this message as part of the commit process:

```
[edit]
jnpr@host1# commit
warning: Hello World!
commit complete
```

Warn If fxp0 Isn't Inherited

A more useful example would be to provide a <xnm:warning> when the fxp0 interface is not being inherited from the re0 or re1 configuration group. Fxp0 is the out-of-band management interface for many Junos devices. The re0 and re1 configuration groups have a unique characteristic in that they are inherited only by the indicated routing engine. So, if a configuration is being committed on routing-engine 0, it ignores the contents of configuration group re1, and vice versa.

The advantage of this behavior is that it allows routing-engine specific configuration to be included within a configuration file that is shared between both routing-engines.

MORE? For more information on the re0 and re1 configuration groups see the *CLI User Guide* within the Junos documentation at www.juniper.net/techpubs/.

One routing-engine specific item that is commonly included within the re0 and re1 groups is the interface fxp0 configuration. The commit script example below tests if

the fxp0 interface is present, and if the interface is present, the script displays a warning message if the configuration is not inherited from either the re0 or re1 configuration group.

```
/* check-fxp0-inheritance.slax */
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 {

   var $fxp0-interface = interfaces/interface[name == "fxp0"];

   /* If fxp0 is configured, but not inherited from re group, then display
warning */
   if( $fxp0-interface &&
      jcs:empty( $fxp0-interface[@junos:group=="re0" || @
junos:group=="re1"] ) ) {
      <xnm:warning> {
         <message> "fxp0 configuration is present but not inherited from re
group";
      }
   }

}
```

Chapter 1 revealed that the post-inheritance candidate configuration provided to commit scripts indicates group inheritance through the `junos:group` attribute. Every configuration element that originates in a group has that group's name tagged to the element as the value of its `junos:group` attribute.

The above commit script takes advantage of this behavior by searching for a `junos:group` attribute on the fxp0 interface node with a name of either re0 or re1. If the fxp0 interface is present, but it lacks a `junos:group` attribute with one of those values, then the following message is displayed when the configuration is committed:

```
[edit]
jnpr@host1# commit
warning: fxp0 configuration is present but not inherited from re group
commit complete
```

Try It Yourself: Host-name Should Inherit from Configuration Group

Create a commit script that generates a commit warning message if the host-name is not inherited from the re0 or re1 configuration groups.

<xnm:warning> Child Elements

The `<message>` child element is required for all `<xnm:warning>` elements. This text is displayed on the same line as the "warning:" statement. While it is often sufficient for simple warnings to include only a `<message>`, at times it is helpful to provide additional information through some of the optional child elements of `<xnm:warning>`. The two most commonly used child elements, which are both described in the following sections, are `<edit-path>` and `<statement>`. These can be seen in typical daemon warning messages such as the following:

```
[edit]
jnpr@host1# commit
[edit interfaces ge-0/0/1 unit 0 family inet]
  'address 10.0.0.1/24'
    warning: identical local address is found on different interfaces
commit complete
```

In this example, the warning message shown is *identical local address is found on different interfaces*, but there are additional lines of information displayed as well. The first line displays the hierarchy where the problem configuration is located, within brackets, and is created by using the <edit-path> element. The second line points out the exact configuration statement that caused the warning. It is created by using the <statement> element and results in a slightly indented string enclosed in single quotes for emphasis.

MORE? To learn about additional child elements of <xnm:warning> see the *Configuration and Diagnostic Automation Guide* within the Junos documentation at www.juniper.net/techpubs/.

<edit-path>

As shown in the prior section, the [edit interfaces ge-0/0/1 unit 0 family inet] output line is an example of <edit-path> output.

This element makes the problem hierarchy stand out and is most useful when there are multiple locations within the configuration that could be the source of the warning. Problems with interface configuration, as demonstrated above, are examples of where the <edit-path> element can be helpful.

The following script displays a warning message when any login class has been assigned the all permission bit. It uses the <edit-path> element to display the hierarchy of the problem login class:

```
/* check-permissions.slax */
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 {
   /* Warn about any login classes with the all permission bit */
   for-each( system/login/class[ permissions == "all" ] ) {
      <xnm:warning> {
         <edit-path> "[edit system login class " _ name _ "]";
         <message> "Permission all is assigned to invalid class.";
      }
   }
}
With this script enabled, assigning any login class the all permission bit
results in the following warning message:
[edit]
jnpr@host1# commit
[edit system login class sandbox]
  warning: Permission all is assigned to invalid class.
commit complete
```

jcs:edit-path

The check-permissions.slax script results in both an <edit-path> and <message> line being displayed within the warning message. But the <edit-path> string was built manually by the script, which can be tedious for deep hierarchies. A more efficient method of including an <edit-path> element is to call the jcs:edit-path template and allow it to automatically generate the <edit-path> element.

One of the default templates included within junos.xsl is jcs:edit-path, which is usable in any commit script. The template works by building the hierarchy of the context node recursively and including the result string within an <edit-path> element. This makes it ideal to use within a for-each loop where each iteration through the loop alters the context node. If the path that should be displayed by <edit-path> is the same as the for-each loop's context node then jcs:edit-path can determine the path by default, but if the context node does not reflect the path that should be included then the $dot parameter of jcs:edit-path can be set to the desired node.

The following script shows how jcs:edit-path can simplify the addition of an <edit-path> to a <xnm:warning> element. This script is designed to provide a warning message about any logical interface that does not have a description configured:

```
/* check-descriptions.slax */
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 {

   /* Loop through all logical interfaces */
   for-each( interfaces/interface/unit ) {

      /* Missing description */
      if( jcs:empty( description ) ) {
         <xnm:warning> {
            call jcs:edit-path();
            <message> "Interface description is missing.";
         }
      }
   }
}
```

The check-descriptions.slax script results in warning messages similar to this when the interface descriptions are missing:

```
[edit]
jnpr@host1# commit
[edit interfaces interface ge-0/0/3 unit 0]
  warning: Interface description is missing.
[edit interfaces interface ge-0/1/0 unit 0]
  warning: Interface description is missing.
commit complete
```

Using the jcs:edit-path template causes the full hierarchy of the context node to be displayed above the warning message. This makes it clear to the user exactly what interface and unit needs to be corrected to remove the warning.

Try It Yourself: ISIS Interface Lacks Family Iso

Create a warning message for every interface enabled for the ISIS protocol that does not have family iso configured. Include an <edit-path> to better document the problem.

<statement>

The <statement> element indicates the exact configuration statement that caused the warning. Returning to the daemon warning message shown earlier in the chapter, the 'address 10.0.0.1/24' line is the statement of the warning message:

```
[edit]
jnpr@host1# commit
[edit interfaces ge-0/0/1 unit 0 family inet]
  'address 10.0.0.1/24'
   warning: identical local address is found on different interfaces
commit complete
```

The statement can be specified manually, similar to the <edit-path> element, by including the <statement> element with its appropriate value within the <xnm:warning> element. In addition, the jcs:statement template can be called to automatically generate a <statement> element based on the context node, or the template's $dot parameter can be set to select a different node.

This example script provides a warning for event scripts that are referenced within an event policy, but are not enabled within the configuration. It includes both an <edit-path> and a <statement> within its <xnm:warning> elements:

```
/* check-event-scripts.slax */
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 {

   /* Save these hierarchy nodes */
   var $event-options = event-options;
   var $system = system;

   /* Look for any referenced event scripts */
   for-each( event-options/policy/then/event-script ) {

      /* Record event-script name */
      var $name = name;

      /* Check if enabled in either location */
      var $under-event-options = $event-options/event-script/file[name ==
$name];
      var $under-system-scripts = $system/scripts/op/file[name == $name];

      /* If it isn't enabled in either location then log an warning */
      if( jcs:empty( $under-event-options ) and jcs:empty( $under-system-
scripts ) ) {
         <xnm:warning> {
            call jcs:edit-path( $dot = ancestor::policy );
            call jcs:statement();
            <message> "Event script is not enabled.";
         }
```

```
        }
    }
}
```

This script uses the `jcs:edit-path` and `jcs:statement` templates to easily create `<edit-path>` and `<statement>` elements based on the context node or a separate reference node. The `<statement>` should refer to the event-script configuration statement so the `jcs:statement` template can use the context node of the `for-each` loop. The `<edit-path>` however, should be set to the event policy that triggers the event script. This is accomplished by setting the `$dot` parameter to the grandparent event policy by using the ancestor axis in the location path.

Here is the full warning message that is displayed when an event script is not properly enabled. It provides all the information required to identify exactly what needs to be resolved:

```
[edit]
jnpr@host1# commit
[edit event-options policy save-core-files]
  'event-script save-core-files.slax;'
    warning: Event script is not enabled.
commit complete
```

<syslog>

While the `<xnm:warning>` element writes a warning message to the console of the committing user, the `<syslog>` element can be used to generate more permanent warnings by writing them to the syslog.

The `<syslog>` element has a single `<message>` child element that contains the actual text to be logged to the syslog. All syslog messages generated by this element are sent from the daemon facility with a severe warning.

NOTE No syslog message is generated by the `<syslog>` element in the following two circumstances:

- When a `commit check` is being performed.
- During the initial boot-up commit.

The following script demonstrates how to use the `<syslog>` element to write warning messages to the syslog:

```
/* check-loopback-filter.slax */
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 {
    var $lo0-interface = interfaces/interface[name == "lo0"];
    if( jcs:empty( $lo0-interface/unit[name=="0"]/family/inet/filter/input
) ) {
        <syslog> {
            <message> "Warning: no lo0.0 firewall filter is assigned.";
        }
    }
}
```

When no firewall filter is configured for the lo0.0 interface the following message is sent to the syslog:

```
Nov 19 22:02:43  host1 cscript: Warning: no lo0.0 firewall filter is
assigned.
```

Comparison to jcs:syslog()

The <syslog> result tree element for commit scripts performs a role similar to the jcs:syslog() function, which is available to all script types, in that the <syslog> element causes its text to be written to the syslog. One difference between the two approaches, though, is that unlike the jcs:syslog() function, the <syslog> result tree element has no control over its facility and severity; it always logs messages from the daemon facility at warning severity. In other words, the <syslog> message:

```
<syslog> "This is a syslog message";
```

Is equivalent to:

```
expr jcs:syslog( "daemon.warning", "This is a syslog message" );
```

Other differences are a result of when the instruction is processed by Junos. The jcs:syslog() function logs the message to the syslog immediately, while the script is still executing, but the <syslog> element is only evaluated when the commit script result trees are examined by the Junos management daemon following the completion of all commit scripts. This has two implications:

- Syslog messages from <syslog> are only logged if the commit process is successfully completed. A <xnm:error> element in the result tree causes the syslog message to not be logged. The jcs:syslog() function is always logged, whether the commit results in an error or not.
- Syslog messages from jcs:syslog() are logged before messages from <syslog>. This affects their order within the syslog message file even if the <syslog> elements occur earlier in the commit script than the jcs:syslog() function.

Other differences exist, all of which are shown in Table 10.1:

Table 10.1 Comparison of <syslog> and jcs:syslog()

	<syslog>	jcs:syslog()
Supported script types	Commit scripts	Commit, Op, and Event scripts
Facility / Severity	Daemon / warning	Configurable
Commit halted by <xnm:error>	Message is not logged	Message is logged
"commit check" performed	Message is not logged	Message is logged
When logged	After all commit scripts finish processing	During commit script processing
Boot up commit	Message is not logged	Message is not logged

Try it Yourself: Compare Syslog Methods

Create a commit script that logs two syslog messages, one using <syslog> and the other using jcs:syslog(). Compare the syslog results when a commit is performed versus a commit check.

<xnm:error>

So far this chapter has explored how to provide feedback from the commit process in the form of warning messages displayed on the console or sent to the syslog. But in addition to giving feedback, commit scripts can take control of the commit process itself. If a script finds a fatal flaw within the candidate configuration the script can halt the commit process, preventing application of the undesired configuration. This control is achieved by using the <xnm:error> result tree element.

The <xnm:error> element is used in the same way as <xnm:warning> and has identical child elements. Like <xnm:warning> the <xnm:error> element requires a <message> child element that contains the text to display to the committing user. But unlike a <xnm:warning> message, a <xnm:error> message halts the commit process. Warning messages remind users to fix small flaws or draw attention to parts of the configuration that need to be added. If the configuration should never be permitted to commit in its current form, errors are used. This situation could occur because of catastrophic configuration lapses, such as a completely missing hierarchy, or as the result of more subtle problems that violate an organization's policies.

Basic Sanity Checking

The simplest use for <xnm:error> is to do basic sanity checking. As mentioned in the last section, this might consist of ensuring that certain hierarchies are always present, such as the [edit interfaces] or [edit protocols] hierarchy. Or it might dig deeper, requiring that a re0 and re1 configuration group be configured, or that ospf is enabled on all core interfaces.

Consider the following example that requires that the SSH service is enabled, that the 'jnpr' account is configured, and that the fxp0 interface has been assigned an IP address:

```
/* basic-sanity-check.slax */
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 {

   /* Ensure that ssh is enabled */
   if( jcs:empty( system/services/ssh ) ) {
      <xnm:error> {
         <message> "SSH must be enabled.";
      }
   }

   /* Ensure that user account jnpr exists */
   if( jcs:empty( system/login/user[name == "jnpr"] ) ) {
      <xnm:error> {
```

```
            <message> "The jnpr user account must be created.";
        }
    }

    /* Verify that fxp0 has an IP address */
    var $fxp0-interface = interfaces/interface[name == "fxp0"];
    if( jcs:empty( $fxp0-interface/unit[name=="0"]/family/inet/address/name
) ) {
        <xnm:error> {
            <message> "fxp0 must have an IP address.";
        }
    }
}
```

In the basic-sanity-check.slax script, three separate tests are performed and a <xnm:error> element is written to the result tree if any elements fail. A single <xnm:error> element is sufficient to halt the commit, but in the worst case scenario all three errors could occur:

```
[edit]
jnpr@host1# commit
error: SSH must be enabled.
error: The jnpr user account must be created.
error: fxp0 must have an IP address.
error: 3 errors reported by commit scripts
error: commit script failure
```

Try It Yourself: Sanity Checking

Write a commit script that generates a <xnm:error> if the [edit system], [edit interfaces], or [edit protocols] hierarchies are missing.

<edit-path> and <statement>

All child elements supported by <xnm:warning> are supported by <xnm:error> as well, including the <edit-path> and <statement> elements. Also, the jcs:edit-path and jcs:statement templates can be used in an identical manner to the <xnm:warning> element.

The following script demonstrates the use of the jcs:edit-path template with a <xnm:error> message. It checks for any EBGP peers that do not have a configured prefix-limit and requires they be fixed before the commit can succeed:

```
/* check-ebgp-peers.slax */
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 {

    /* Retrieve AS Number */
    var $asn = routing-options/autonomous-system/as-number;

    /* Scroll through all EBGP peers */
    for-each( protocols/bgp/group[ peer-as != $asn ]/neighbor ) {

        if( jcs:empty( family/inet/unicast/prefix-limit/maximum ) &&
```

```
            jcs:empty( ../family/inet/unicast/prefix-limit/maximum ) ) {
            <xnm:error> {
               call jcs:edit-path();
               <message> "EBGP peer must have prefix limit defined.";
            }
         }
      }
}
```

The check-ebgp-peers.slax script assumes that the `peer-as` is defined at the BGP group rather than at the neighbor level, and that an autonomous-system has been defined within `[edit routing-options]`. (These two assumptions could also have been verified by the commit script itself). The script verifies that all EBGP peers have a prefix-limit defined at either the neighbor or group level. If any peers lack the limit the script displays an error message including the `<edit-path>` displaying the neighbor hierarchy, and halts the commit:

```
[edit]
jnpr@host1# commit
[edit protocols bgp group AS65535 neighbor 10.0.0.2]
  EBGP peer must have prefix limit defined.
error: 1 error reported by commit scripts
error: commit script failure
```

Try It Yourself: Incorrect Autonomous-system Number

Write a commit script that generates a `<xnm:error>` if the autonomous-system number is not set to `65000`. Include `<edit-path>` and `<statement>` elements to better document the problem.

Errors Based on Chassis Components

Commit scripts have access to the same Junos API elements as op and event scripts, which means that they can be programmed to generate `<xnm:error>` messages based on the configuration as well as on other factors such as what PICs are installed, etc.

But perform such programming with caution. Information from the API is not always available and commit scripts need to be programmed to handle this possible problem correctly. For example, during the boot-up commit many API elements do not return data, as the relevant daemons have not yet initialized. Also remember that commit scripts run on all routing-engines of the system, and many times only the master routing-engine has access to all the information available from the API. This makes it very possible for a commit script that requests API information and is running on one routing-engine to reach a different decision than the same commit script running on another routing-engine, which can lead to inconsistency across the routing-engines. In addition, PICs are added, FPCs are removed, and other changes occur, all outside of the commit process. Event scripts can be used to watch for these changes, but remember that commit scripts do not see the changes until the next commit is requested.

Based on the above risks, it is usually best to have commit scripts avoid using information from Junos API elements when making their configuration control decisions.

Feedback and Control Options

This chapter has presented numerous examples of commit scripts that provide feedback or control the configuration through the <xnm:warning>, <syslog>, and <xnm:error> result tree elements. Table 10.2 shows a list of the different example conditions and actions taken:

Table10.2 Examples from This Chapter

Condition	Action
Fxp0 isn't inherited	<xnm:warning> message
'all' permission bit assigned to class	<xnm:warning> message
Missing interface description	<xnm:warning> message
Event script is not enabled	<xnm:warning> message
Loopback firewall filter is missing	<syslog> message
SSH is not enabled	<xnm:error> message and halt commit
'jnpr' user account is missing	<xnm:error> message and halt commit
Fxp0 does not have an IPv4 address	<xnm:error> message and halt commit
EBGP peer lacks prefix-limit	<xnm:error> message and halt commit

Consider the examples that were used, the conditions that were checked, and the resulting actions. Is it worthwhile for a commit script to look for these conditions? Do the actions seem appropriate? The answer to these questions varies for each network because all network administrators have different opinions about what should and should not be present within their configuration. Individual administrators will all make different decisions on which conditions are problematic enough for warning messages and which configuration flaws should result in commit errors.

This is what makes Junos commit scripts so useful – they allow administrators from each network to decide for themselves what warnings are delivered and what configuration problems might cause commit errors. Commit scripts provide the tools necessary to automate the commit process to run in a way that works best for each individual network.

Try It Yourself: Brainstorm Warnings and Errors

Write a list of configuration problems that would be of interest for your network. Note if they should result in a <xnm:warning>, <syslog>, or <xnm:error>.

BEST PRACTICE Creating feedback and control options is an ongoing process. Many configuration violations are readily apparent, but other problems might only be realized after a human error has resulted in a disruption. When correcting configuration errors, always consider if the addition of new commit script checks could prevent the issue from occurring again.

Element and Template Summary

Tables 10.3 and 10.4 summarize the configuration feedback and control result tree elements and templates.

Table 10.3 Feedback and Control Result Tree Elements

Result Tree Elements	Child Elements
<xnm:warning> - Display warning message to console	<message> - [Required] – The text to display <edit-path> - Relevant configuration hierarchy of the warning message <statement> - Configuration statement that is the cause of the warning message
<syslog> - Writes a message to the syslog from the daemon facility with a severity of warning	<message> - [Required] – The text to write to the syslog
<xnm:error> - Display error message to console and halt commit process	<message> - [Required] – The text to display <edit-path> - Relevant configuration hierarchy of the error <statement> - Configuration statement that is the cause of the error

Table 10.4 Feedback and Control Templates

Templates	Template Parameters
jcs:edit-path - Generates an <edit-path> element automatically for the context node	$dot - Generate <edit-path> for selected node rather than context node
jcs:statement- Generates a <statement> element automatically for the context node	$dot - Generate <statement> for selected node rather than context node

Chapter 11

Changing the Configuration

Chapter 10 demonstrated the use of <xnm:warning>, <syslog>, and <xnm:error> to provide configuration feedback and control. Those techniques can be used to bring problems to the attention of the committing user, who can then resolve them manually.

But many configuration issues do not require manual intervention. The ability to automate configuration changes is a powerful capability of commit scripts. This means that instead of asking the user to fix an issue, the script simply resolves the problem automatically, ensuring that the configuration is structured according to the network's policies and desires.

Adding/Editing/Replacing

Configuration changes are made by adding a <change> element to the result tree. The <change> element encloses the configuration hierarchy and the statement that should be changed:

```
<change> {
    <system> {
        <host-name> "host2";
    }
}
```

Notice that the full hierarchy is enclosed, starting at the top-level. The configuration mode `load` command is used the same way, by including the full hierarchy of the desired change. In the <change> element above, if `host-name` is present in the configuration then it is changed to `"host2"`, otherwise `host-name` is added to the configuration with `"host2"` as its value.

MORE? To learn more about the `load` configuration mode command see the *CLI User Guide* within the Junos documentation at www.juniper.net/techpubs/.

As previously shown in Figure 9.1, configuration changes affect the candidate configuration. This is the same configuration that the user has edited and requested to be committed. After the commit process is completed, all changes performed by commit scripts are included within the committed configuration, just as if they had been done manually by a user.

The fact that <change> elements impact the candidate configuration is made explicit by using the `commit check` command. This performs a commit to check for errors, but does not actually apply the committed configuration. Yet the <change> elements are still applied to the candidate configuration. For example, when the above <change> element is loaded into a commit script and a `commit check` is performed, the following effect is seen:

```
[edit]
jnpr@host1# show system host-name
host-name host1;

[edit]
jnpr@host1# commit check
configuration check succeeds

[edit]
jnpr@host1# show system host-name
host-name host2;
```

Notice after the `commit check` is performed that the host-name within the configuration is now set to host2, yet the prompt still says host1. This is because the committed configuration has not changed, only the candidate configuration has changed. In the normal commit process this difference is unnoticed because a successful commit results in the candidate configuration becoming the committed configuration, but because `commit check` does not follow the complete commit process it allows the effects of `<change>` elements to be clearly shown.

Try It Yourself: Commit Check and the <change> Element

Write a simple commit script that changes a single configuration setting. Perform a `commit check` and verify that the candidate configuration is altered, but that the committed configuration remains unchanged. Perform a normal `commit` and verify that the change is now visible in the committed configuration.

Automatically Add Missing Configuration

Chapter 10 included an example script called basic-sanity-check.slax that checked for the following three conditions and halted the commit with an error if they were missing:

- SSH service is enabled
- `"jnpr"` user account is present
- Fxp0 interface has an IPv4 address

Providing an error message and asking the user to fix the problems is a valid strategy, but if the missing configuration is standardized then why not have the commit script fix the problem automatically? It might not be possible to generate the fxp0 interface automatically, because the address is different for every device. Still, enabling SSH and adding the "jnpr" user account should be standardized, and this action could be performed by the commit script instead of requiring manual intervention.

The following commit script modifies the basic-sanity-check.slax script and automatically fixes the standardized portions:

```
/* basic-sanity-check-and-fix.slax */
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 {

   /* Ensure that ssh is enabled - if not then enable it */
   if( jcs:empty( system/services/ssh ) ) {
      <change> {
         <system> {
            <services> {
               <ssh>;
            }
         }
      }
      <xnm:warning> {
```

```
            <message> "Enabling ssh";
        }
    }

    /* Ensure that user account jnpr exists - if not then add it */
    if( jcs:empty( system/login/user[name == "jnpr"] ) ) {
        <change> {
            <system> {
                <login> {
                    <user> {
                        <name> "jnpr";
                        <class> "super-user";
                        <authentication> {
                            <encrypted-password> "$1$RHL6So3y$kcOy3vb6YiWf6FAJzHi
7j1";
                        }

                    }
                }
            }
        }
        <xnm:warning> {
            <message> "Adding jnpr user account";
        }
    }

    /* Verify that fxp0 has an IP address */
    var $fxp0-interface = interfaces/interface[name == "fxp0"];
    if( jcs:empty( $fxp0-interface/unit[name=="0"]/family/inet/address/name
) ) {
        <xnm:error> {
            <message> "fxp0 must have an IP address.";
        }
    }
}
```

The modified script still checks for the same problems, but now when SSH is not enabled or the `jnpr` account is missing it adds a `<change>` element to the result tree and fixes the problem automatically. The `<xnm:error>` element has been changed to a `<xnm:warning>` element to notify the committing user that the script made automated configuration changes.

BEST PRACTICE Include a `<xnm:warning>` element when making automated changes to notify the user that the script has changed the configuration.

The check of the fxp0 interface remains the same as the original script. If no IPv4 address is configured then a `<xnm:error>` element is generated, which halts the commit and requires manual intervention. It would be possible to hardcode the needed address into the script and fix the problem automatically, but that would result in a separate script per each Junos device. Alternatively, if the fxp0 address is assigned in a deterministic fashion based on some value that the script can access (such as the host-name), then it would be possible for the script to generate it automatically. In the case of the script above, that condition still remains an error and requires the user's attention to fix.

Try It Yourself: Automated Configuration Fixes

Identify a standard part of your configuration that should always be present. Write a commit script that automatically adds it when missing and generates a `<xnm:warning>` message informing the user of the change.

Replacing Configuration

Using the <change> element is equivalent to using the `load replace` configuration command. By default, configuration content is merged into the configuration, new items are added, and conflicting statements are overridden. But as with the `load replace` configuration mode command, it is possible to indicate that the enclosed configuration should replace the existing configuration, rather than simply merging into it. This is done by adding the `replace` attribute to the desired configuration element with a value of `"replace"`.

MORE? To learn more about the `load replace` configuration mode command see the *CLI User Guide* within the Junos documentation at www.juniper.net/techpubs/.

Consider the case of a Junos device that should always have its syslog messages file configured in the following manner:

```
file messages {
   any notice;
}
```

No other facilities should be configured and no other syslog options should be included. If a commit script is configured to enforce this configuration by checking for incorrect configuration statements under the messages file and issuing a <change> element, without using the `replace="replace"` attribute, then the desired result is not achieved.

Here is the initial faulty commit script:

```
/* faulty-check-syslog-messages.slax */
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 {

   var $messages-file = system/syslog/file[name == "messages"];
   if( jcs:empty( $messages-file ) || count( $messages-file/* ) > 2 ||
      jcs:empty( $messages-file/contents[ name == "any" ]/notice ) ) {

      <change> {
         <system> {
            <syslog> {
               <file> {
                  <name> "messages";
                  <contents> {
                     <name> "any";
                     <notice>;
                  }
               }
            }
         }
      }
      <xnm:warning> {
         <message> "Syslog messages file configuration corrected";
      }
   }
```

The script correctly catches an improperly configured messages file, but it does not change the configuration correctly. With the following configuration pre-commit:

```
file messages {
    any any;
    daemon verbose;
}
```

The configuration is changed to the following by the commit script:

```
file messages {
    any notice;
    daemon verbose;
}
```

The any facility is correctly changed to use the notice severity, but the daemon facility is not removed. This is because the change was merged into the existing configuration; it did not replace it. So any existing configuration statements that were not overridden by merged statements remain.

In order to replace the existing file messages hierarchy, the `replace` attribute is added with a value of `"replace"`. Here is the correct commit script:

```
/* check-syslog-messages.slax */
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 {

    var $messages-file = system/syslog/file[name == "messages"];
    if( jcs:empty( $messages-file ) || count( $messages-file/* ) > 2 ||
        jcs:empty( $messages-file/contents[ name == "any" ]/notice ) ) {

        <change> {
            <system> {
                <syslog> {
                    <file replace="replace"> {
                        <name> "messages";
                        <contents> {
                            <name> "any";
                            <notice>;
                        }
                    }
                }
            }
        }
        <xnm:warning> {
            <message> "Syslog messages file configuration corrected";
        }
    }
}
```

Placing the `replace` attribute on a configuration element causes that element and all its descendents to be replaced within the configuration by the new configuration. If the messages file configuration is the following prior to the commit:

```
file messages {
   any any;
   daemon verbose;
}
```

It becomes the following after the commit script has replaced the existing file messages hierarchy:

```
file messages {
   any notice;
}
```

Try It Yourself: Replacing Configuration Hierarchies

Create a commit script that enforces the requirement that the ospf configuration should consist solely of an assignment of all interfaces into area 0.

jcs:emit-change

As the prior section mentioned, the <change> element must contain the complete hierarchy of the configuration statement. For changes that are only one level deep this is not an issue, but it can become unwieldy when the change is deep within a hierarchy.

Consider the change required to add an ingress firewall filter to the loopback interface:

```
<change> {
   <interfaces> {
      <interface> {
         <name> "lo0";
         <unit> {
            <name> "0";
            <family> {
               <inet> {
                  <filter> {
                     <input> {
                        <filter-name> "ingress";
                     }
                  }
               }
            }
         }
      }
   }
}
```

The verbosity of deep configuration changes can at times be avoided by using the `jcs:emit-change` template, rather than by manually building <change> elements with the full parent hierarchy. The `jcs:emit-change` template creates a <change> element within which it builds the hierarchy of the context node and then includes the specified change at that hierarchy level. Similar to the `jcs:edit-path` and `jcs:statement` templates the `jcs:emit-change` template uses the current context node, or allows an alternate node to be specified through its `$dot` parameter. The `$content` parameter is required and contains the change that should be applied to the context node hierarchy. The final parameter that can be used is the `$tag` parameter which allows the default <change> element to be substituted with a <transient-change> . Transient changes are discussed at the end of this chapter.

Consider the following script that uses the `jcs:emit-change` template to assign a firewall filter to the lo0 interface:

```
/* add-loopback-filter.slax */
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 {

   var $lo0-interface = interfaces/interface[name=="lo0"]/unit[name=="0"];
   if( jcs:empty( $lo0-interface/family/inet/filter/input[filter-name ==
"in"] ) ) {
      /* Create the change */
      var $change = {
         <filter> {
            <input> {
               <filter-name> "in";
            }
         }
      }
      call jcs:emit-change( $dot = $lo0-interface/family/inet, $content =
$change );
      <xnm:warning> {
         <message> "Adding lo0 input filter.";
      }
   }
}
```

The add-loopback-filter.slax script assigns an input firewall filter for lo0.0 if not already assigned. The code takes advantage of the `$dot` parameter to minimize the amount of hierarchy that must be included within the configuration change. Because the `$dot` is set at the family inet level, the code change can be crafted for that hierarchy and can begin with a reference to the `<filter>` element. With the `$dot` parameter set, and the `$content` parameter assigned to the configuration change, the `jcs:emit-change` template can create the necessary `<change>` element with the needed configuration hierarchy for the desired change.

ALERT! The `$dot` parameter must refer to an existing node within the configuration. With the above script, if the lo0 interface is not currently configured, or lacks a unit 0 with family inet, then the `jcs:emit-change` template generates a commit error:

```
error: jcs:emit-change called with invalid location (dot)
```

The add-loopback-filter.slax script manually creates a `<xnm:warning>` message to indicate that the configuration has been changed, but that is not necessary when using the `jcs:emit-change` template. The template has a `$message` parameter that can automatically create a `<xnm:warning>` message to display to the committing user. The following lines of code:

```
call jcs:emit-change( $dot = $lo0-interface/family/inet, $content = $change
);
<xnm:warning> {
   <message> "Adding lo0 input filter.";
}
Could have been written instead as:

var $message = "Adding lo0 input filter.";
call jcs:emit-change($dot = $lo0-interface/family/inet, $content = $change,
$message);
```

Try It Yourself: Family MPLS on LDP Interfaces

Create a commit script that calls the `jcs:emit-change` template to add family mpls to every interface, configured under `[edit protocols ldp]`, that lacks it.

Deleting

At this point, the ability to add, edit, and replace configuration has been demonstrated, but it is also possible to delete configuration statements as well as complete configuration hierarchies. This action is accomplished by including the `delete` attribute on the configuration element that should be removed inside the `<change>` element, and setting the attribute's value to `"delete"`.

For example, a configured time-zone could be removed by using this `<change>` element:

```
<change> {
    <system> {
        <time-zone delete="delete">;
    }
}
```

The `delete` attribute can also delete complete configuration hierarchies. This example removes the `[edit protocols ospf traceoptions]` configuration hierarchy:

```
<change> {
    <protocols> {
        <ospf> {
            <traceoptions delete="delete">;
        }
    }
}
```

Configuration hierarchies with identifiers can be deleted by including the `delete` attribute on the configuration element and also including its identifier element as a child element (typically `<name>`). This `<change>` element removes the ge-0/0/0 interface from the configuration. The actual configuration element is `<interface>`, but its child element `<name>` identifies the specific interface configuration to remove:

```
<change> {
    <interfaces> {
        <interface delete="delete"> {
            <name> "ge-0/0/0";
        }
    }
}
```

Some configuration statements can have multiple values. To delete a single value from a multiple-value statement, include the `delete` attribute for the statement's configuration element and also include the statement's value. For example, login classes can define multiple permission bits. The following `<change>` element deletes only the `"all"` permission bit from the `"admin"` login class:

```
<change> {
    <system> {
        <login> {
            <class> {
                <name> "admin";
                <permissions delete="delete"> "all";
```

```
            }
        }
    }
}
```

The following example shows the ospf-fxp0-disabled.slax script. This commit script ensures that the fxp0 interface is included under area 0 as a disabled interface and not under any other areas:

```
/* ospf-fxp0-disabled.slax */
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 {

    /* Loop through all ospf areas */
    for-each( protocols/ospf/area ) {

        /* Area 0, verify that fxp0 is disabled */
        if( name == "0.0.0.0" && jcs:empty( interface[name=="fxp0.0"]/disable
) ) {
            /* Add the statement since it's missing */
            var $content = {
                <interface> {
                    <name> "fxp0.0";
                    <disable>;
                }
            }
            var $message = "Adding fxp0.0 disable";
            call jcs:emit-change( $content, $message );

        }
        /* All other areas, fxp0 not allowed */
        else if( name != "0.0.0.0" && interface[name=="fxp0.0"] ){
            var $content = {
                <interface delete="delete"> {
                    <name> "fxp0.0";
                }
            }
            var $message = "Removing fxp0.0";
            call jcs:emit-change( $content, $message);
        }
    }
}
```

The commit script loops through all the ospf areas. If the current area is 0.0.0.0 then the script checks if fxp0 is included as a disabled interface. If it is not present in disabled form, then the `jcs:emit-change` template is used to generate the needed `<change>` element to add the desired configuration. No `$dot` needs to be set because the context node is the ospf area and the `$content` change is designed for that hierarchy level.

For other areas, the commit script checks if fxp0 is configured, and if it is the script uses `jcs:emit-change` to create a `<change>` element that deletes it.

Assume the configuration prior to commit is the following:

```
[edit]
jnpr@host1# show protocols ospf
area 0.0.0.0 {
    interface lo0.0;
    interface ge-0/0/0.0;
}
area 0.0.0.1 {
    interface ge-0/0/1.0;
    interface fxp0.0;
}
```

The following warning messages are displayed to the committing user:

```
[edit]
jnpr@host1# commit
[edit protocols ospf area 0.0.0.0]
  warning: Adding fxp0.0 disable
[edit protocols ospf area 0.0.0.1]
  warning: Removing fxp0.0
commit complete
```

And the ospf configuration is successfully changed:

```
[edit]
jnpr@host1# show protocols ospf
area 0.0.0.0 {
    interface lo0.0;
    interface ge-0/0/0.0;
    interface fxp0.0 {
        disable;
    }
}
area 0.0.0.1 {
    interface ge-0/0/1.0;
}
```

NOTE As the output example above shows, using the `$message` parameter of `jcs:emit-change` to display a warning message automatically includes an `<edit-path>` with the context node used as the hierarchy level. This is why `[edit protocols ospf area 0.0.0.0]` and `[edit protocols ospf area 0.0.0.1]` are included in the output.

Try It Yourself: Deleting Invalid Name-servers

Create a commit script for an organization whose name-servers all fall within the 10.0.1.0/24 subnet. Delete any configured name-servers from outside that subnet.

Activating/Deactivating

A commit script can activate or deactivate configuration by following a similar method as deletion. To activate configuration, add the `active` attribute with a value of `"active"`. To deactivate configuration, add the `inactive` attribute with a value of `"inactive"`.

The same approach is used for each type of configuration element with the delete attribute. For example, to deactivate the `[edit snmp]` hierarchy, use the following `<change>` element:

```
<change> {
    <snmp inactive="inactive">;
}
```

Or, to activate the "jnpr" user account, the following <change> element can be used:

```
<change> {
    <system> {
        <login> {
            <user active="active"> {
                <name> "jnpr";
            }
        }
    }
}
```

Deactivating

Deactivating improper configuration is an alternative to generating a <xnm:error> and rejecting the entire commit request. Recall the check-ebgp-peers.slax script from Chapter 10 that generated an error if any EBGP peers lacked a prefix-limit. The following commit script takes a different approach, instead of failing the commit it deactivates the problem EBGP peer and generates a warning message. The peer is not allowed to be active until the prefix-limit is put in place, but this problem does not prevent other configuration changes from taking effect.

```
/* deactivate-ebgp-peers.slax */
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 {

    /* Retrieve AS Number */
    var $asn = routing-options/autonomous-system/as-number;

    /* Scroll through all EBGP peers */
    for-each( protocols/bgp/group[ peer-as != $asn ]/neighbor ) {

        if( jcs:empty( family/inet/unicast/prefix-limit/maximum ) &&
            jcs:empty( ../family/inet/unicast/prefix-limit/maximum ) ) {
            var $content = {
                <neighbor inactive="inactive"> {
                    <name> name;
                }
            }
            var $message = "EBGP peer is missing prefix limit.  Deactivating.";
            call jcs:emit-change( $dot = .., $content, $message );
        }
    }
}
```

The EBGP neighbor is deactivated by setting the inactive attribute to "inactive" for the neighbor element and enclosing the <name> identifier element. The $dot parameter for jcs:emit-change is set to allow the configuration change to be based on the parent BGP group hierarchy rather than the current <neighbor> context node. A message is provided to the jcs:emit-change template so the committing user sees a warning similar to the following:

```
[edit]
```

```
jnpr@jhost1# commit
[edit protocols bgp group EBGP neighbor 10.0.0.2]
  warning: EBGP peer is missing prefix limit.  Deactivating.
commit complete
```

Activating

Activating configuration is done in the same way as deactivating configuration, except that the `active` attribute is added, rather than the `inactive attribute`, and its value is set to `"active"`. The difficult part of this process is knowing whether or not the configuration needs to be activated, because inactive configuration is not present within the post-inheritance configuration provided to the commit script.

It is possible to retrieve the configuration, including inactive statements, by using the `<get-configuration>` Junos API element, which returns the pre-inheritance candidate configuration by default. Any inactive elements within this configuration have an `inactive` attribute with a value of `"inactive"`.

The following shows an example of a commit script that retrieves the candidate configuration to detect if the autonomous-system is deactivated, and if it is inactive the script generates the necessary `<change>` element to activate it:

```
/* activate-asn.slax */
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 {

   /* Retrieve the current configuration, when doing this be aware of PR
452398 */
   var $configuration = jcs:invoke( "get-configuration" );

   if( $configuration/routing-options/autonomous-system[@inactive] ) {
      <change> {
         <routing-options> {
            <autonomous-system active="active">;
         }
      }
      <xnm:warning> {
         <message> "Activating ASN configuration.";
      }
   }
}
```

ALERT! Do not use the `<get-configuration>` API element within a commit script prior to the following releases: 9.3S5, 9.4R4, 9.5R3, 9.6R2, or 10.0R1, as it can cause the Junos device to hang while booting. For further details refer to PR 452398.

Reordering

While the order of most configuration statements is unimportant, some parts of the configuration, such as firewall terms, are processed sequentially and must be in the proper order. Assume that the following routing-policy has been configured:

```
policy-statement accept-ospf {
```

```
    term reject {
        then reject;
    }
    term ospf {
        from protocol ospf;
        then accept;
    }
}
```

The intention of the policy is to allow OSPF routes and reject all others, but the faulty term order causes all routes to be rejected. To correct this the ospf term must be inserted prior to the reject term. In configuration mode the following command can be used:

```
[edit policy-options policy-statement accept-ospf]
jnpr@host1# insert term ospf before term reject
```

Following this command, the policy now reads:

```
policy-statement accept-ospf {
    term ospf {
        from protocol ospf;
        then accept;
    }
    term reject {
        then reject;
    }
}
```

Commit scripts are also capable of moving elements before or after their siblings within the configuration. To move configuration hierarchies, such as the different policy terms, use the following format:

```
<element insert="after | before" identifier-element="reference-element-
name"> {
    <identifier-element> "moving-element-name";
}
```

The configuration element is referenced by including its unique identifying element (typically called name):

```
<term> {
    <name> "ospf";
}
```

The `insert` attribute is used to indicate that an insertion needs to be performed, and the attribute is set to `"before"` or `"after"` to show where it should be placed in reference to the other element.

```
<term insert="before"> {
    <name> "ospf";
}
```

Finally, the reference element is identified by including the identifier element name as an attribute (in the above example, this is `"name"`) with the reference element's value:

```
<term insert="before" name="reject"> {
    <name> "ospf";
}
```

The full `<change>` element to accomplish this is the following:

```
<change> {
   <policy-options> {
      <policy-statement> {
         <name> "accept-ospf";
         <term insert="before" name="reject"> {
            <name> "ospf";
         }
      }
   }
}
```

Reordering among multiple values for a single configuration statement differs slightly from the above examples because the various elements have no identifying child elements, and instead differ only in their assigned values. In this case, the format used is the following:

```
<element insert="after | before" name="reference-value"> "moving-value";
```

Consider the various static-route next-hops:

```
<routing-options> {
   <static> {
      <route> {
         <name> "192.168.1.0/24";
         <next-hop> "10.0.0.1";
         <next-hop> "10.0.0.2";
      }
   }
}
```

To cause the 10.0.0.2 next-hop to be moved in front of the 10.0.0.1 next-hop the following code can be used:

```
<change> {
   <routing-options> {
      <static> {
         <route> {
            <name> "192.168.1.0/24";
            <next-hop insert="before" name="10.0.0.1"> "10.0.0.2";
         }
      }
   }
}
```

MORE? For more information on reordering configuration elements see the Junoscript API Guide within the Junos documentation at www.juniper.net/techpubs/.

One common reason to reorder is if a commit script is adding firewall terms, policy terms, or other configuration elements that must then be placed within their proper order. Unless the entire parent hierarchy is being replaced, the new elements are appended to the existing content. In many cases this placement is incorrect and must be fixed through proper reordering. For example, the following script adds a "block-private-ranges" policy to all EBGP peers that lack it, and ensures that the added policy is placed at the beginning of the import policy chain.

```
/* add-block-privates.slax */
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 {

   /* Retrieve AS Number */
   var $asn = routing-options/autonomous-system/as-number;

   /* Scroll through all EBGP groups */
   for-each( protocols/bgp/group[ peer-as != $asn ] ) {
      if( jcs:empty( import[1][. == "block-private-ranges" ] ) ) {
         var $content = {
            /* Add and insert at beginning */
            if( count( import ) > 0 ) {
               var $current-first = import[1];
               <import insert="before" name=$current-first> "block-
private-ranges";
            }
            /* Just add */
            else {
               <import> "block-private-ranges";
            }
         }
         var $message = "Adding block-private-ranges import policy.";
         call jcs:emit-change( $content, $message );
      }
   }
}
```

The add-block-private-ranges.slax script iterates through every EBGP group and verifies that the `"block-private-ranges"` policy is defined as the first policy. If not then it adds the policy, and if other policies are present it reorders the policy in the same step.

Try It Yourself: Reorder Firewall Terms

Create a commit script that adds a term to a firewall filter, if missing, and then inserts it at the beginning of the filter.

Renaming

The final type of configuration change is renaming. This change is the same action performed by using the `rename` command in configuration mode:

```
[edit protocols bgp]
jnpr@host1# rename group AS65535 to group AS65495
```

To rename an element, use the `rename` attribute with the following format:

```
<element rename="rename" name="new name"> {
   <identifier-element> "old name";
}
```

For example, renaming a BGP group could be done like this:

```
<change> {
   <protocols> {
      <bgp> {
         <group rename="rename" name="AS65495"> {
            <name> "AS65535";
         }
      }
   }
}
```

The above code renames the existing BGP group AS65535 to a new name of AS65495.

The following script shows one application of the commit script renaming capability. The commit script convert-to-hyphens.slax is intended to enforce a hypothetical organization's policy that all prefix-lists use hyphens rather than underscores within their names.

```
/* convert-to-hyphens.slax */
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 {

   /* Loop through all prefix-lists */
   for-each( policy-options/prefix-list ) {

      /* Do they have an underscore in their name? */
      if( contains( name, "_" ) ) {

         /* Translate _ to - */
         var $new-name = translate( name, "_", "-" );
         var $content = {
            <prefix-list rename="rename" name=$new-name> {
               <name> name;
            }
         }
         var $message = "Translating _ to -";
         call jcs:emit-change( $dot=., $content, $message );
      }
   }
}
```

This initial configuration:

```
[edit policy-options]
jnpr@host1# show
prefix-list CUST_A {
   10.0.0.0/24;
   10.0.1.0/24;
}
prefix-list CUST_B {
   192.168.100.0/24;
   192.168.200.0/24;
}
```

Results in the following warning messages at commit:

```
[edit]
jnpr@jhost1# commit
[edit policy-options prefix-list CUST_A]
  warning: Translating _ to -
[edit policy-options prefix-list CUST_B]
  warning: Translating _ to -
commit complete
```

And the configuration is changed:

```
[edit policy-options]
jnpr@host1# show
prefix-list CUST-A {
    10.0.0.0/24;
    10.0.1.0/24;
}
prefix-list CUST-B {
    192.168.100.0/24;
    192.168.200.0/24;
}
```

The convert-to-hyphens.slax commit script works by looping through all prefix-lists and selecting the names that contain underscores. These name strings are converted through the `translate()` function from underscore to hyphen and then the `jcs:emit-change` template is used to generate a`<change>` element with the appropriate renaming instructions.

However, this script is flawed in that while the prefix-lists are changed, their references in policy-statements and firewall filters are not altered. Without correction, this results in a commit error when the referenced prefix-list is not found during the commit process. The following *Try it Yourself* example resolves this issue. The Appendix can be consulted to see the solution to the exercise.

Try It Yourself: Modify Convert-to-hyphens.slax

Modify the convert-to-hyphens.slax commit script. Along with renaming the prefix-list, the references to the prefix-list in policy-statements and firewall filters should also be set to the new name.

Transient Changes

Up to this point only persistent configuration changes have been displayed. Persistent changes are done using the `<change>` result tree element. They directly impact the candidate configuration, and are included within the normal configuration just like manual configuration changes.

However, in addition to persistent changes commit scripts also have the ability to perform transient changes, which are changes that affect the checkout configuration applied to the Junos daemons but never show up in the normal configuration file.

Transient changes have the following characteristics:

- Created by the `<transient-change>` result tree element.
- Require `allow-transients` to be configured under `[edit system scripts commit]`.
- Do not affect the candidate configuration, only affect the checkout configuration.
- Do not appear in the committed configuration file.
- Recreated by commit scripts during each commit. To remove a transient change, remove the commit script.

In summary, a transient change is a configuration change that is only visible to Junos, not to the users. Transient changes are used primarily in combination with configuration macros, as discussed in Chapter 12, but are also useful if there are large configuration statements that should be hidden from the configuration file. An example of this could be large user authentication keys.

Consider the following user account configuration:

```
user jnpr {
    uid 2000;
    class super-user;
    authentication {
        ssh-dsa "ssh-dss
                        VlAYXzZ5XUDmBwAGgARS4ILMlhU2ozpfSePZmMqfqsvMCeSsssYtTX7W1DEnbvA+SdWg35zhS4
                        utAYnlAjzJtaqoB4EYmk8xt5DCeNd/vSwTMOhlsXFXYHkxOnO5Va5+etQ1c3j9d0WoO7+Mu6yx
                        zgJnBN6I9lLYK8jbAAAAFQCkjYEHTB8PnKkXUBf2yk+aykSeaQAAAIAe2I7x9TYC9Eas1BqMgZ
                        bOBGgXrOjo/a5ZJdFIY22in2t9yAhaqbVbgSpPN9lIDtOab1JG3bzb8Gb9OpvKBiOtMKj4vd8f
                        hUm5SzujJW7sP+FkWixevi+EnfUFQRIgLTeKKe6QDAPxOUcH84pWKMuxiW9xlcXAJzvuGb2iQQ
                        BNLwAAAIAE2tJjK+dJZWoudzvv8pDWWk2H+QxzEGpsCWJQJNVAarY1nCgy5+pbXyX7M9I1FC/
                        fjmaCBwZR//JuYRfo+29LTsCMAk9bOfSrToszXvXgtJ86nWzn1Sz9w3yDgtxpoD8R/mUqa8Xf5
                        J7uGwOT6ypBMa+7u2sGrqD6RiSvCGxGbQ== example"; ## SECRET-DATA
    }
}
```

Rather than allowing this key to clutter up the configuration, it is possible to embed it within a commit script, and have the script transiently add the key to the configuration at commit time. Here is an example of how this can be done:

```
/* add-user-key.slax */
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 {
    <transient-change> {
        <system> {
            <login> {
                <user> {
                    <name> "jnpr";
                    <authentication> {
                        <ssh-dsa> {
                            <name> "ssh-dss AAAAB3NzaC1kc3MAAACBAM5Yu7v/VlAYXzZ5" _
                                  "XUDmBwAGgARS4ILMlhU2ozpfSePZmMqfqsvMCeSsssYt" _
                                  "TX7W1DEnbvA+SdWg35zhS4utAYnlAjzJtaqoB4EYmk8x" _
                                  "t5DCeNd/vSwTMOhlsXFXYHkxOnO5Va5+etQ1c3j9dOWo" _
                                  "O7+Mu6yxzgJnBN6I9lLYK8jbAAAAFQCkjYEHTB8PnKkX" _
                                  "UBf2yk+aykSeaQAAAIAe2I7x9TYC9Eas1BqMgZbOBGgX" _
                                  "rOjo/a5ZJdFIY22in2t9yAhaqbVbgSpPN9lIDtOab1JG" _
                                  "3bzb8Gb9OpvKBiOtMKj4vd8fhUm5SzujJW7sP+FkWixe" _
                                  "vi+EnfUFQRIgLTeKKe6QDAPxOUcH84pWKMuxiW9xlcXA" _
                                  "JzvuGb2iQQBNLwAAAIAE2tJjK+dJZWoudzvv8pDWWk2H" _
                                  "+QxzEGpsCWJQJNVAarY1nCgy5+pbXyX7M9I1FC/fjmaC" _
                                  "BwZR//JuYRfo+29LTsCMAk9bOfSrToszXvXgtJ86nWzn" _
                                  "1Sz9w3yDgtxpoD8R/mUqa8Xf5J7uGwOT6ypBMa+7u2sG" _
                                  "rqD6RiSvCGxGbQ== example";
                        }
                    }
```

```
            }
         }
      }
   }
}
```

The add-user-key.slax script adds the 'jnpr' user's ssh dsa key transiently to the configuration. To enable it, the allow-transients configuration statement must be configured:

```
system {
   scripts {
      commit {
         allow-transients;
         file add-user-key.slax;
      }
   }
}
```

With the above script in place, the 'jnpr' user account requires only the following configuration because its authentication key is added transiently:

```
user jnpr {
   uid 2000;
   class super-user;
}
```

jcs:emit-change

The jcs:emit-change template can be used to generate <transient-change> elements in the same way as <change> elements. The difference is that its $tag parameter must be set to "transient-change" as its default behavior is to generate a <change> element.

```
call jcs:emit-change( $content, $tag = "transient-change" );
```

Viewing Transient Changes

While transient configuration changes do not appear in the normal configuration file, it is possible to view them by adding | display commit-scripts to the show command in configuration mode or the show configuration command in operational mode.

For example, the full 'jnpr' user account configuration can be seen along with the effects of the add-user-key.slax script in this manner:

```
[edit]
jnpr@host1# show system login | display commit-scripts
user jnpr {
   uid 2000;
   class super-user;
   authentication {
      ssh-dsa "ssh-dss
                  AAAAB3NzaC1kc3MAAACBAM5YVlAYXzZ5XUDmBwAGgARS4ILMlhU2ozpfSePZmMqfqsvMCeSsssY
                  tTX7W1DEnbvA+SdWg35zhS4utAYnlAjzJtaqoB4EYmk8xt5DCeNd/vSwTMOhlsXFXYHkxOnO5Va
                  5+etQ1c3j9dOWoO7+Mu6yxzgJnBN6I9lLYK8jbAAAAFQCkjYEHTB8PnKkXUBf2yk+aykSeaQAAA
                  IAe2I7x9TYC9Eas1BqMgZb0BGgXr0jo/a5ZJdFIY22in2t9yAhaqbVbgSpPN9lIDtOab1JG3bz-
                  b8Gb9OpvKBiOtMKj4vd8fhUm5SzujJW7sP+FkWixevi+EnfUFQRIgLTeKKe6QDAPxOUcH84pWKM
```

```
                    uxiW9xlcXAJzvuGb2iQQBNLwAAAIAE2tJjK+dJZWoudzvv8pDWWk2H+QxzEGpsCWJQJNVAarY1
                    nCgy5+pbXyX7M9I1FC/fjmaCBwZR//JuYRfo+29LTsCMAk9bOfSrToszXvXgtJ86nWzn1Sz9w3
                    yDgtxpoD8R/mUqa8Xf5J7uGwOT6ypBMa+7u2sGrqD6RiSvCGxGbQ== example"; ## SE-
                    CRET-DATA
   }
}
```

ALERT! At the time of this writing, using the `replace` attribute within a `<transient-change>` does not work correctly and the change is merged rather than replaced. Viewing the output of `show configuration | display commit-scripts` makes it appear that the replace operation was successful, but it does not accurately reflect the committed configuration in this scenario.

Try It Yourself: Transient Root Authentication Key

Create a commit script that adds the root authentication key transiently to the configuration. Use the `jcs:emit-change` template to do so.

Element and Template Summary

Table 11.1, 11.2, and 11.3 summarize the configuration change result tree elements, configuration element attributes, and change templates.

Table 11.1 Configuration Change Result Tree Elements

Result Tree Elements	Configuration Affected
`<change>` - perform a persistent configuration change	`Candidate configuration`
`<transient-change>` - perform a transient configuration change	`Checkout configuration`

Table 11.2 Configuration Change Attributes

Configuration Element Attributes	Effect
`active="active"`	Activates the configuration statement or hierarchy
`delete="delete"`	Deletes the configuration statement or hierarchy
`inactive="inactive"`	Deactivates the configuration statement or hierarchy
`insert="before | after"`	Inserts the configuration statement or hierarchy before or after the referenced statement or hierarchy
`rename="rename"`	Renames the configuration statement or hierarchy
`replace="replace"`	Replaces the existing configuration with the replacement configuration

Table 11.3 Configuration Change Template Parameters

Template	Template Parameters
`jcs:emit-change` – Generates a `<change>` or `<transient-change>` element	`$dot` – Change the current hierarchy for the configuration change `$content` – [Required] - The desired configuration change, relative to the current hierarchy `$tag` – Generate a `'change'` (default) element or a `'transient-change'` element

Chapter 12

Configuration Macros

The last three chapters have shown how commit scripts can automate configurations by providing feedback, halting the commit of invalid configurations, and performing configuration changes. But all commit script logic shown so far has been based on the contents of the configuration itself. This is often sufficient, but commit scripts can be even more flexible by adding configuration macros as scripts that can then work with arbitrary data instead of relying on only the standard configuration syntax.

Overview

Configuration macros contain arbitrary information that can be embedded within a configuration. The name *macro* can become somewhat misleading because the macros are actually just data, stored within the configuration. The actual programming logic comes entirely from commit scripts, which can react to the data within the configuration macros.

Configuring Macros

Macros are configured by using the `apply-macro` configuration statement:

```
[edit]
jnpr@host1# set apply-macro example
```

The above configuration command adds a macro named "example" at the top-level of the configuration hierarchy. Macros can be added to any configuration hierarchy and multiple macros can be present at the same hierarchy as long as they have different names:

```
interfaces{
   lo0 {
      apply-macro macro1;
      apply-macro macro2;
   }
}
```

ALERT! The `apply-macro` command does not show up in the command help and cannot be auto-completed. It is not a hidden command; the reason it doesn't appear in the command help is to prevent all the different `apply-*` statements from cluttering up the help of every hierarchy level. The same result is seen with the `apply-flag` command. In both cases, although they are supported configuration statements, the full command must be typed out and is not shown in the help output.

Some macros only consist of their name, but it is also possible to configure macro parameters. Each parameter must have a name and can optionally have a value as well. Multiple parameters can be configured but each parameter for a particular macro must have a unique name:

```
apply-macro customer-1 {
   interface ge-0/0/0.0;
   protocol bgp;
   service-level gold;
}
apply-macro customer-2 {
   interface ge-2/0/1.0;
   protocol static;
   service-level silver;
}
```

Working with Macros

Configuration macros can be retrieved from or added to the configuration in the same way as any other configuration element. Here are the two customer macros that were shown in the past section, this time shown in XML format:

```
<apply-macro> {
    <name> "customer-1";
    <data> {
        <name> "interface";
        <value> "ge-0/0/0.0";
    }
    <data> {
        <name> "protocol";
        <value> "bgp";
    }
    <data> {
        <name> "service-level";
        <value> "gold";
    }
}
<apply-macro> {
    <name> "customer-2";
    <data> {
        <name> "interface";
        <value> "ge-2/0/1.0";
    }
    <data> {
        <name> "protocol";
        <value> "static";
    }
    <data> {
        <name> "service-level";
        <value> "silver";
    }
}
```

Each macro is identified by its <name> element. And each parameter within a macro is represented by a <data> element with the name and value, if a value is present, stored in its <name> and <value> child elements.

Assuming these values are stored at the top-level of the configuration, they could be retrieved using the following code:

- Retrieve the interface value of the customer-1 macro:

```
var $interface = apply-macro[name == "customer-1"]/data[name=="interface"]/
value;
```

- Loop through all protocol parameters of macros that start with "customer":

```
for-each( apply-macro[ starts-with( name, "customer")]/
data[name=="protocol"] ) {
    ....
}
```

- The following example shows the <change> element necessary to add a "time-interval" macro to the ge-0/0/0 interface stanza:

```
<change> {
    <interfaces> {
        <interface> {
```

```
            <name> "ge-0/0/0";
            <apply-macro> {
               <name> "time-interval";
               <data> {
                  <name> "start-time";
                  <value> "08:00";
               }
               <data> {
                  <name> "stop-time";
                  <value> "13:00";
               }
               <data> {
                  <name> "default-down";
               }
            }
         }
      }
}
```

The above <change> element adds the following apply-macro to the configuration:

```
apply-macro time-interval {
   default-down;
   start-time 08:00;
   stop-time 13:00;
}
```

Macro Uses

Adding macros to the configuration has no default impact because Junos ignores all macros and their configured parameters. It is only through Junos automation scripts that macros gain their significance.

Using configuration macros is entirely arbitrary and up to the script writer, but they are typically used for the following:

- Data storage
- Instruction sets
- Exception flags
- Custom configuration syntax

Each of these possibilities is discussed in the remaining sections of this chapter.

Data Storage

The first use for configuration macros is arbitrary data storage. In this case the macros exist solely as a convenient location to store information within the configuration. This use extends beyond commit scripts, as op scripts or event scripts might have reason to retain information in the configuration as well.

Anything of interest can be placed within a configuration macro as long as it fits in the parameter name and value format. The following script shows an example of how configuration macros can be used to store unique information within the configuration. In this case, the script stores the date when each EBGP peer was added to the configuration, as well as the user that added it:

```
/* record-bgp-commit-info.slax */
```

```
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";

var $macro-name = "commit-info";

match configuration {

   /* Retrieve AS Number */
   var $asn = routing-options/autonomous-system/as-number;

   /* Grab the date */
   var $date = substring-before( $localtime_iso, " " );

   /* Loop through each EBGP peer */
   for-each( protocols/bgp/group[ peer-as != $asn ]/neighbor ) {

      /* Are they missing their commit-info macro? */
      if( jcs:empty( apply-macro[name == $macro-name] ) ) {

         /* Add the apply-macro to it */
         var $content = {
            <apply-macro> {
               <name> $macro-name;
               <data> {
                  <name> "date";
                  <value> $date;
               }
               <data> {
                  <name> "user";
                  <value> $user;
               }
            }
         }
         var $message = "Adding commit information";
         call jcs:emit-change( $content, $message );
      }
   }
}
```

The first time each EBGP peer is added to the configuration the commit script notices that it lacks the "commit-info" configuration macro and generates the configuration change necessary to add the macro with the current date and the committing user.

As a result, the configuration contains additional information about the creation of each peer, which can be used in any way desired:

```
[edit protocols bgp]
jnpr@host1# show
group AS65535 {
   peer-as 65535;
   neighbor 10.0.0.1 {
      apply-macro commit-info {
         date 2009-11-25;
         user jnpr;
      }
   }
```

```
    neighbor 10.0.0.2 {
        apply-macro commit-info {
            date 2009-11-10;
            user roy;
        }
    }
}
```

Instruction Set

The second use of a configuration macro is causing a commit script to perform specific actions and then remove the macro. The presence of the macro at a certain hierarchy of the configuration might be enough information for the commit script to perform its duties, or the macro might contain one or more parameters that provide additional information about what tasks the commit script should perform.

The following commit script shows an example of a configuration macro applied for this purpose. Whenever the configuration statement `apply-macro set-core-interface` is applied to an interface the following configuration changes are made by the script:

- Add family mpls to interface
- Add family iso to interface
- Enable interface for MPLS protocol
- Enable interface for LDP protocol
- Enable interface for ISIS protocol
- Remove macro

```
/* set-core-interface.slax */
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";

var $macro-name = "commit-info";

match configuration {

    /* Look for instruction macro */
    for-each( interfaces/interface/unit[ apply-macro/name == "set-core-
interface"] ) {

        /* Add families and remove macro */
        var $content = {
            <apply-macro delete="delete"> {
                <name> "set-core-interface";
            }
            <family> {
                <iso>;
                <mpls>;
            }
        }
        var $message = "Setting as core interface...";
        call jcs:emit-change( $content, $message );
```

```
        /* Assemble interface name */
        var $name = ../name _ "." _ name;

        /* Add to protocols */
        <change> {
            <protocols> {
                <mpls> {
                    <interface> {
                        <name> $name;
                    }
                }
                <ldp> {
                    <interface> {
                        <name> $name;
                    }
                }
                <isis> {
                    <interface> {
                        <name> $name;
                    }
                }
            }
        }
    }
}
```

The macro is added to the desired interface prior to the commit:

```
[edit interfaces ge-1/0/0 unit 0]
jnpr@host1# show
apply-macro set-core-interface;
family inet {
    address 10.0.1.1/24;
}
```

And following the commit, the macro has been removed and the required changes have been put in place for the interface to function as a core interface:

```
[edit interfaces ge-1/0/0]
jnpr@host1# show
unit 0 {
    family inet {
        address 10.0.1.1/24;
    }
    family iso;
    family mpls;
}

[edit]
jnpr@host1# show protocols
mpls {
    interface ge-1/0/0.0;
}
isis {
    interface ge-1/0/0.0;
}
ldp {
    interface ge-1/0/0.0;
}
```

Try It Yourself: MTU Changes

Design a configuration macro with two parameters. The first parameter refers to the desired MTU value and the second is a regular expression for all interfaces that should be assigned the MTU value. Create a commit script that looks for the configuration macro in the [edit interfaces] hierarchy and makes the instructed MTU changes in response. The configuration macro should be removed as part of the configuration change.

Exception Flag

The third use of configuration macros is an exception flag macro, which is used to indicate that a configuration section should be skipped by commit scripts. This macro is useful when a commit script looks at all occurrences of a particular configuration, such as all interfaces, and performs the same operation on all of them by default.

Using an example of a commit script that modifies interface configuration, the script could verify the absence of a particular macro prior to changing the interface. That way, if the device's administrators do not want a subset of their interfaces to be modified by the script they can set the macro on those interfaces, which then flags them as being an exception to the normal commit script processing.

Recall the check-ebgp-peers.slax commit script shown in Chapter 10 that displayed a `<xnm:error>` if any EBGP peers lacked a prefix-limit. This script has now been modified to use the macro `"skip-prefix-limit-check"` as an exception flag:

```
/* check-ebgp-peers-with-exception.slax */
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 {

    /* Retrieve AS Number */
    var $asn = routing-options/autonomous-system/as-number;

    /*
     * Scroll through all EBGP peers - do not worry about peers with
     * "skip-prefix-limit-check" macro applied.
     */
    for-each( protocols/bgp/group[ peer-as != $asn ]/neighbor ) {

        if( jcs:empty( family/inet/unicast/prefix-limit/maximum ) &&
            jcs:empty( ../family/inet/unicast/prefix-limit/maximum ) &&
            jcs:empty( apply-macro[name == "skip-prefix-limit-check"] ) ) {
            <xnm:error> {
                call jcs:edit-path();
                <message> "EBGP peer must have prefix limit defined.";
            }
        }
    }
}
```

With the modified version of this commit script in place, the following configuration is allowed by the commit script because the macro defined for neighbor 10.0.0.2

causes the script to ignore its missing prefix list:

```
bgp {
   group AS65535 {
      peer-as 65535;
      neighbor 10.0.0.1 {
         family inet {
            unicast {
               prefix-limit {
                  maximum 10000;
               }
            }
         }
      }
      neighbor 10.0.0.2 {
         apply-macro skip-prefix-limit-check;
      }
   }
}
```

Custom Configuration Syntax

The fourth and final use for macros is creating custom configuration syntax by pairing `apply-macro` statements with transient configuration changes.

The `apply-macro` statements remain permanently in the configuration, and from the point of view of the users they are the actual configuration statements. But the commit script translates the macros into their equivalent Junos commands and provides the true Junos configuration transiently to the daemons.

This is a powerful capability that allows configurations to be standardized as well as simplified. The standardization comes because the full configuration expansion is automated, which prevents human error. And the simplification comes from the amount of standardized configuration that can be hidden from normal view.

Provisioning automation is a typical use of this type of configuration macro. By abstracting the configuration details into a simple interface of name and value pairs, commit scripts that work with these macros can ensure the provisioned configuration can be accurately and consistently generated.

Here is an example that performs the common task of sharing interface routes with both the inet.0 and inet.2 routing tables, but accomplishes all the tasks with a single `apply-macro both-ribs` statement.

The configuration required:

```
routing-options {
   apply-macro both-ribs;
   static {
      route 172.0.0.0/8 next-hop 172.25.45.1;
   }
}
```

The expand-both-ribs.slax commit script:

```
/* expand-both-ribs.slax */
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 both-ribs macro is present, then expand the configuration
transiently */
   if( routing-options/apply-macro[ name == "both-ribs" ] ) {
      <transient-change> {
         <routing-options> {
            <interface-routes> {
               <rib-group> {
                  <inet> "both-ribs";
               }
            }
            <rib-groups> {
               <name> "both-ribs";
               <import-rib> "inet.0";
               <import-rib> "inet.2";
            }
         }
      }
   }
}
```

The checkout configuration provided to Junos daemons:

```
routing-options {
   apply-macro both-ribs;
   interface-routes {
      rib-group inet both-ribs;
   }
   static {
      route 172.0.0.0/8 next-hop 172.25.45.1;
   }
   rib-groups {
      both-ribs {
         import-rib [ inet.0 inet.2 ];
      }
   }
}
```

Note the difference between the configuration as seen by the user, and the actual configuration that is committed and provided to Junos daemons. The visible configuration is smaller because the standardized portions of the expanded configuration do not need to be viewed. From a user's perspective, the `apply-macro both-ribs` statement creates a both-ribs rib-group and shares the interface routes with both inet.0 and inet.2.

Try It Yourself: Custom Firewall Filter

Design a configuration macro that has two parameters, one that indicates the control protocol between PE and CE (BGP, OSPF, etc.), and the other that indicates the policer bandwidth. Create a commit script that transiently creates a firewall filter for each logical interface with that macro configured. The firewall filter should allow all packets from the control protocol in the first term, and allow all packets in the second term, but rate-limit them to the bandwidth specified in the macro.

MORE? Find another example of custom configuration syntax in the Appendices.

Appendices

Appendix A: Supplemental Junos Automation Information from Part One

This Appendix supplements the information previously discussed in Part One by providing five additional op scripts as well as example solutions to the Try It Yourself sections.

Op Script Examples

This first section of the Appendix provides five additional scripts, which highlight the possibilities that op scripts provide and makes use of the lessons learned in this volume. Extensive comments are included within the scripts to provide documentation on their structure.

Display Time

In this first example, the `display-time` template alters the earlier example of Chapter 3 by providing an easier way for the user to select the ISO format or the normal format.

In the prior example the `desired-format` command-line argument was added, allowing the user to choose what format in which to display the time. When the `display-time` template is called the `$format` parameter is set to the value of the `$desired-format` global parameter.

There is an easier way to write the script to let the user choose the format. Remember that global parameters and variables are accessible in the entire script, not just in the main template. This means that it is not necessary to pass the `$format` value to the `display-time` template as a template parameter. Instead, the `display-time` template can simply use the global parameter.

The script operates in the same manner as the script in Chapter 3, but in this case the named template accesses the global parameter instead of relying on the main template to pass the format as a template parameter:

```
/* show-time.slax */
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";

/* This is imported into the Junos CLI help text */
var $arguments = {
  <argument> {
    <name> "display-format";
    <description> "Choose either iso or normal";
  }
}

/* Command-line argument */
param $display-format;

match / {
  <op-script-results> {

    /* Call the display-time template */
```

```
    call display-time;

  }
}

/* Output the localtime to the console in either iso (default) or normal
format */
template display-time {
  if( $display-format == "iso" ) {
    <output> "The iso time is" _ $localtime_iso;
  }
  else {
    <output> "The time is" _ $localtime;
  }
}
```

The output is the same as provided by the example of Chapter 3:

```
user@Junos> op show-time display-format iso
The iso time is 2009-05-12 21:01:10 PDT

user@Junos> op show-time display-format normal
The time is Tue May 12 21:01:13 2009
```

Static route

The next script is used to add a static route. First the `jcs:get-input()` function is used to learn the static route and then it is used again to learn the next-hop. Once these two values are known the script instructs configuration changes, then loads and commits the change:

```
/* add-route.slax */
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> {

    /* Ask for the static route */
    var $static-route = jcs:get-input("Enter the static route: ");

    /* Ask for the next-hop */
    var $next-hop = jcs:get-input("Enter the next-hop: ");

    /* Create the configuration change */
    var $configuration = <configuration> {
              <routing-options> {
                <static> {
                  <route> {
                    <name> $static-route;
                    <next-hop> $next-hop;
                  }
                }
              }
            }

    /* Open a connection */
    var $connection = jcs:open();

    /* Call jcs:load-configuration and provide the connection and
```

```
configuration
    * change to make.
    */
   var $results := { call jcs:load-configuration( $connection,
$configuration ); }

   /* Check for errors - report them if they occurred */
   if( $results//xnm:error ) {
     for-each( $results//xnm:error ) {
       <output> message;
     }
   }

   /* If there are no errors then report success */
   if( jcs:empty( $results//xnm:error ) ) {
     <output> "Committed without errors.";
   }

   /* Close the connection */
   var $close-results = jcs:close($connection);
 }
}
```

Here is an example of the `add-route op` script in operation:

```
user@Junos> op add-route
Enter the static route: 10.6.0.0/16
Enter the next-hop: 192.168.1.4
Committed without errors.
```

And here is the configuration after the scripted change (two routes were already present):

```
user@Junos> show configuration routing-options
static {
  route 10.1.0.0/16 next-hop 192.168.1.1;
  route 10.2.0.0/16 next-hop 192.168.1.1;
  route 10.6.0.0/16 next-hop 192.168.1.4;
}
autonomous-system 65500;
```

show-bgp-policy

This is an example of a custom show command that was created to enhance Junos operation. The goal of this script is to provide an easy way to peruse the complete policy chain of a BGP peer. With multiple BGP policies in an import or export policy chain, it can become troublesome to determine exactly when or if a prefix is accepted/rejected as the policies typically do not appear in the desired order within the `policy-options` configuration.

This script extracts the policy chain for the selected peer in the import or export direction. It then retrieves the configuration text for each policy, and displays it on the console in sequential order. Now a user only has to execute the op script and he can then read the complete policy chain from start to finish.

This shows the output of the script when asked to report the import policies for peer 10.0.0.1:

```
user@Junos> op show-bgp-policy neighbor 10.0.0.1 direction import
```

```
BGP Neighbor: 10.0.0.1 in group EBGP
Import Policies: block-private set-local-pref accept-by-community
Policy: block-private
  policy-statement block-private {
    from {
      route-filter 192.168.0.0/16 orlonger;
      route-filter 10.0.0.0/8 orlonger;
      route-filter 172.16.0.0/12 orlonger;
    }
    then reject;
  }
Policy: set-local-pref
  policy-statement set-local-pref {
    term default {
      then {
        local-preference 50;
      }
    }
    term pref-75 {
      from community pref-75;
      then {
        local-preference 75;
      }
    }
    term pref-100 {
      from community pref-100;
      then {
        local-preference 100;
      }
    }
  }
Policy: accept-by-community
  policy-statement accept-by-community {
    term accept {
      from community from-64500;
      then accept;
    }
    term reject {
      then reject;
    }
  }
```

An additional feature of this script is that a user can use the `database` command-line argument to select between the committed configuration and the candidate configuration. The committed configuration is used by default. However, the option to view the policy chain in the candidate configuration is useful to verify the BGP policy configuration prior to committing any policy changes.

The script code for `show-bgp-policy.slax` follows:

```
/*
 * This op script displays the full chain of policy configuration for a given
 * neighbor and import/export direction. Either the candidate or the
committed
 * database can be used, with the committed database used by default.
 *
 * Usage: op show-bgp-policy neighbor 10.0.0.1 direction import
 *
 */

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";

/*
 * The $arguments global variable is a special variable which Junos reads to
build
 * the CLI help for the script. The command-line arguments will appear
within the help
 * message along with their description if the following format is followed.
 */
var $arguments = {
  <argument> {
    <name> "neighbor";
    <description> "Required: The BGP neighbor address";
  }
  <argument> {
    <name> "direction";
    <description> "Required: Policy direction, either import or export";
  }
  <argument> {
    <name> "database";
    <description> "Optional: Specify either the [committed] or candidate
configuration database";
  }
}

/* These global parameters are assigned based on their corresponding
command-line arguments */
param $neighbor;
param $direction;
param $database="committed";

match / {
  <op-script-results> {

    /*
     * The script first sanity checks the $neighbor and $direction command-
line arguments. If they
     * have not been set correctly then the script exits by using the
<xsl:message> command element
     * and specifying that the script should end by setting the terminate
attribute to "yes". Note
     * that <xsl:message> is not a result tree element, it is one of the few
elements that is
     * processed immediately rather than written to the result tree.
     */
    if( jcs:empty( $neighbor ) or jcs:empty( $direction ) or
      ( $direction != "import" and $direction != "export" ) ) {
      <xsl:message terminate="yes"> "The neighbor address and policy
direction must be specified.";
    }

    /*
     * The database should be set to either "committed" or "candidate", if
not then exit the script
     * with an error
     */
    if( $database != "committed" and $database != "candidate" ) {
      <xsl:message terminate="yes"> "The database is not set correctly.";
```

```
    }

    /*
     * This API element is used to retrieve the configuration. Either the
candidate or the committed
     * configuration can be used. The choice is based on the $database
command-line argument. In
     * addition the inherit attribute has been set. This is used to request
that the configuration be
     * retrieved in inherited mode meaning that all configuration from
configuration groups will be
     * inherited into its proper hierarchy level. This is done so that the
script has an accurate view
     * of the current BGP policy configuration.
     */
    var $get-bgp-rpc = <get-configuration database=$database
inherit="inherit"> {
                <configuration> {
                  <protocols> {
                    <bgp>;
                  }
                }
              }

    /*
     * The assembled API element is sent to Junos through jcs:invoke and the
XML response is stored in
     * $bgp-config
     */
    var $bgp-config = jcs:invoke( $get-bgp-rpc );

    /*
     * The BGP neighbor is extracted from the configuration through a
location path. The last()
     * function is used to guarantee that only one element node will be
returned. It returns true
     * only if the node is the last in the node list so only one node can be
selected and assigned to
     * $bgp-neighbor.
     */
    var $bgp-neighbor = $bgp-config/protocols/bgp//neighbor[name ==
$neighbor ][last()];

    /*
     * Error check, if the $bgp-neighbor is missing than jcs:empty() will
return true and the script
     * will be terminated with an error message.
     */
    if( jcs:empty( $bgp-neighbor ) ) {
      <xsl:message terminate="yes"> "BGP Neighbor " _ $neighbor _ " isn't
configured.";
    }

    /*
     * Begin the output. The BGP neighbor will be shown first as well as the
BGP group it is in.
     */
    <output> "BGP Neighbor: " _ $neighbor _ " in group " _ $bgp-neighbor/../
name;

    /*
     * The BGP policy list will now be retrieved. To do this the jcs:first-
of() function is used.
```

```
    * This is necessary because a BGP peer's policy can be configured at up to
three different
    * hierarchy levels: the neighbor level, the group level, or the bgp
level. The peer uses the
    * policy configuration at the most specific level. jcs:first-of() works
by checking multiple
    * node-set arguments. The first node-set that is not empty will be
returned. So in this case
    * three separate location paths are provided (each of which results in a
node-set). The first
    * location path refers to any policies at the neighbor level, the second
pulls policies at the
    * group level, and the last pulls policies at the bgp level. The most
specific location that has
    * policies will be returned and assigned to the $policy-list variable.
    */
   var $policy-list = jcs:first-of( $bgp-neighbor/*[name()==$direction],
                $bgp-neighbor/../*[name()==$direction],
                $bgp-neighbor/../../*[name()==$direction] );

   /*
    * Error check, if there are no policies then the script can terminate.
    */
   if( jcs:empty( $policy-list ) ) {
     <xsl:message terminate="yes"> "There are no " _ $direction _ " policies
for " _ $neighbor;
   }

   /*
    * The policy chain is now output to the console. This is done all within
one line by writing
    * the text strings to a single <output> element through the expr
statement.
    */
   <output> {
     if( $direction == "import" ) {
       expr "Import Policies:";
     }
     else {
       expr "Export Policies:";
     }
     for-each( $policy-list ) {
       expr " " _ .;
     }
   }

   /* A for-each will now loop through each policy individually to allow them
to be displayed */
   for-each( $policy-list ) {

     /* Output the policy name */
     <output> "\nPolicy: " _ .;

     /*
      * The script must retrieve the text version of each policy one by one.
By default the
      * returned configuration is always in XML format. To see the
configuration in text format
      * use the format attribute and set it to "text".
      */
     var $get-policy-rpc = <get-configuration format="text"
database=$database inherit="inherit"> {
               <configuration> {
```

```
                <policy-options> {
                  <policy-statement> {
                    <name> .;
                  }
                }
              }
            }

     /* Send assembled API element to Junos through jcs:invoke(); */
     var $policy-text = jcs:invoke( $get-policy-rpc );

     /*
      * The returned configuration will include the entire hierarchy
including the policy-options
      * statement and enclosing brackets. This is extra clutter that is not
needed so it is removed
      * by using the substring-after and substring functions to remove all
the unnecessary
      * characters. The complete text policy is then output to the console.
      */
     var $cropped-text =substring-after( $policy-text, "policy-options {"
);
     <output> substring( $cropped-text, 1, string-length( $cropped-text )-2
);
    }
  }
}
```

change-password

This next example shows a script that performs an automated configuration change. This script allows a user to self-serve their local account by changing their password from the command-line. Use of this script requires that the user has the necessary permission to make password changes (see Chapter 4). The minimum version required is JUNOS 9.6, as it added the `jcs:get-secret()` function to query for a new password. Here is an example of the output of the script:

```
user@Junos> op change-password
Enter the new password:
Reenter the new password:
Password changed.
```

Here is the script code for `change-password.slax`:

```
/*
 * This op script changes the password for the current user. The password is
 * learned using jcs:get-secret() for security purposes so the minimum Junos
version
 * is 9.6
 */
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> {

    /*
     * Query the user for the new password. Use jcs:get-secret() so that the
```

```
password
    * is not shown on the screen. Ask twice and compare the entries. Terminate
the
    * script if they are not the same.
    */
   var $new-password = jcs:get-secret("Enter the new password: ");
   var $new-password2 = jcs:get-secret("Reenter the new password: ");

   if( $new-password != $new-password2 ) {
     <xsl:message terminate="yes"> "The passwords do not match.";
   }
   if( string-length( $new-password ) == 0 ) {
     <xsl:message terminate="yes"> "The password is blank.";
   }

   /*
    * Assemble the configuration change needed to change the password. Pull
in the
    * user name from the $user default global parameter.
    */
    var $configuration = {
     <configuration> {
       <system> {
         <login> {
           <user> {
             <name> $user;
             <authentication> {
               <plain-text-password-value> $new-password;
             }
           }
         }
       }
     }
   }

   /* Open a connection */
   var $connection = jcs:open();

   /*
    * Send the configuration change and connection to jcs:load-configuration,
it will load
    * the change and commit it.
    */
    var $result := { call jcs:load-configuration($connection,
$configuration); }

    /* Check for commit errors - report them if found */
    if( $result//xnm:error ) {
     <output> jcs:output("Error changing the password");
     for-each( $result//xnm:error ) {
       <output> message;
     }
   }
   else {
     <output> "Password changed.";
   }

   /* Close the connection */
   var $close-results = jcs:close( $connection );
 }
}
```

safe-bgp-clear

This last example shows how scripts can wrap around standard operational mode commands to alter their default behavior. When users enter `clear bgp neighbor` then all BGP peering sessions are restarted. This script makes the command safer by requiring users to confirm that they really do want to restart all their sessions. The minimum version required is Junos 9.6 because the `jcs:get-input()` function is used to perform confirmation.

Here is an example of the output of the script:

```
user@Junos> op safe-bgp-clear peer all
This will clear ALL BGP sessions
Are you sure? (yes/[no]): yes
Cleared 2 connections
```

Here is the script code for `safe-bgp-clear.slax`:

```
/*
 * This script provides a safe version of the "clear bgp neighbor" command.
That command
 * allows an operator to accidently clear all BGP neighbors when no address
is specified.
 * This script requires "peer all" to be specified in order to clear all BGP
neighbors,
 * and it requires confirmation from the operator that they do indeed wish to
clear all
 * of their BGP peers.
 *
 * Minimum Junos version is 9.6 due to the jcs:get-input() function. (This
can be performed
 * in Junos 9.4 and 9.5 by using the deprecated jcs:input() 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";

/*
 * The $arguments global variable is a special variable which Junos reads to
build
 * the CLI help for the script. The command-line arguments will appear
within the help
 * along with their description as long as the following format is followed.
 */
var $arguments = {
  <argument> {
    <name> "peer";
    <description> "Neighbor address to drop or 'all' for all sessions";
  }
}
/* This global parameter will have its value set based on the matching
command-line argument */
param $peer;

match / {
  <op-script-results> {
```

```
    /*
     * The script requires that a peer be specified. If the user did not enter
a peer on the
     * command-line then display an error and exit.
     */
    if( string-length( $peer ) == 0 ) {
      <xsl:message terminate="yes"> "You must specify which BGP peer you want
to clear.";
    }

    /*
     * If peer 'all' was entered on the command-line then the user will need to
confirm the choice
     * before clearing all the BGP peers.
     */
    if( $peer == "all" ) {

      /*
       * Request confirmation from user before clearing all the BGP peers. In
Junos 9.4 & 9.5
       * jcs:input() can be used instead.
       */
      var $prompt = "This will clear ALL BGP sessions\nAre you sure? (yes/
[no]): ";
      var $response = jcs:get-input( $prompt );

      if( $response == "yes" ) {
        /*
         * The user confirmed they wish to clear all sessions so call the
execute-command
         * template.
         */
        call execute-command();
      }
      else {
        /*
         * If they decided not to clear all the sessions then display that
choice to the
         * console.
         */
        <output> "Clear all cancelled";
      }
    }
    else {
      /* There is no need to confirm the clearing of a specific peer, just
execute the clear
       * command by calling the execute-command template.
       */
      call execute-command();
    }
  }
}

/*
 * This template will invoke the clear bgp neighbor command and display any
output to the console.
 */
template execute-command() {

  /* There is no mapped API Element for clear bgp neighbor so the <command>
element will be used */
  var $command = {
    /* If all is specified then just do the normal command, otherwise include
```

```
the peer address */
   if( $peer == "all" ) {
     <command> "clear bgp neighbor";
   }
   else {
     <command> "clear bgp neighbor " _ $peer;
   }
  }

  /* Execute the command and retrieve the results */
  var $results = jcs:invoke( $command );

  /* Copy output to the result tree so that it will be displayed on the
console */
  copy-of $results;
}
```

Try It Yourself Sample Solutions

This section of the Appendix provides sample solutions for each of the Try It Yourself sections in Chapters 1 through 4.

Chapter 1: Try It Yourself: Viewing Junos Configuration in XML

Show the following configuration hierarchy levels in XML on a Junos device:

```
(e.g. show configuration system | display xml)

[system]
[interfaces]
[protocols]


user@Junos> show configuration system | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.0R4/junos">
  <configuration junos:commit-seconds="1248125252" junos:commit-
localtime="2009-07-20 14:27:32 PDT" junos:commit-user="user">
     <system>
       <host-name>Junos</host-name>
       <login>
         <user>
           <name>user</name>
           <authentication>
             <encrypted-password>pVFST7cOl4Hu2</encrypted-password>
           </authentication>
         </user>
       </login>
     </system>
  </configuration>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>

user@Junos> show configuration interfaces | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.0R4/junos">
  <configuration junos:commit-seconds="1248125420" junos:commit-
localtime="2009-07-20 14:30:20 PDT" junos:commit-user="user">
     <interfaces>
       <interface>
         <name>lo0</name>
```

```
        <unit>
          <name>0</name>
          <family>
            <inet>
              <address>
                <name>10.3.3.3/32</name>
              </address>
            </inet>
            <iso>
              <address>
                <name>47.0000.3333.3333.3333.00</name>
              </address>
            </iso>
          </family>
        </unit>
      </interface>
    </interfaces>
  </configuration>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>

user@Junos> show configuration protocols | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.0R4/junos">
  <configuration junos:commit-seconds="1248125504" junos:commit-
localtime="2009-07-20 14:31:44 PDT" junos:commit-user="user">
     <protocols>
       <ospf>
         <area>
           <name>0.0.0.0</name>
           <interface>
             <name>ge-1/0/0.0</name>
           </interface>
         </area>
       </ospf>
       <pim>
         <interface>
           <name>ge-1/0/0.0</name>
         </interface>
       </pim>
     </protocols>
  </configuration>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

Chapter 1:
Try It Yourself: Writing XML in the SLAX Abbreviated Format

Rewrite the following configuration using the SLAX abbreviated XML format:

```
system {
  host-name r1;
  login {
    message "Unauthorized access prohibited.";
  }
}

<system> {
  <host-name> "r1";
```

```
  <login> {
    <message> "Unauthorized access prohibited.";
  }
}
```

Chapter 2: Try It Yourself: Adding Comments to the Hello World Script

Make the following modifications to the Hello World script:

1. Add a multi-line comment at the beginning that describes the purpose of the script.

2. Add an additional comment before the `<output> "Hello World!";` line which states that the script is writing to the console.

After the two modifications have been made, replace the prior version of hello-world.slax on your Junos device with the new version. Execute the script again and verify that the new comments did not change the operation.

```
/*
 * hello-world.slax - This script will output "Hello World!" to the console.
 */
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> {
    /* This writes the string to the console. */
    <output> "Hello World!";
  }
}
```

Chapter 2: Try It Yourself: Adding Additional Output to the Hello World Script

Modify the Hello World script once again. This time, add two additional lines of output to the console above the `Hello World!` string.

Replace the prior version of hello-world.slax on your Junos device with the changed version. Execute the script again and see the affect the new <output> elements have on the script output.

```
/*
 * hello-world.slax - This script will output "Hello World!" to the console.
 */
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> {
    /* This writes the string to the console. */
    <output> "I have a message for you.";
```

```
    <output> "Here is the message:";
    <output> "Hello World!";
  }
}
```

Chapter 2: Try It Yourself: Writing Your Own Script Using the Boilerplate

Using the configuration boilerplate, create a new op script that outputs three separate lines of text to the console. Copy this script to your Junos device and enable it. Now you can verify it by executing it from the command-line.

```
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> "One";
    <output> "Two";
    <output> "Three";
  }
}
```

Chapter 3: Try It Yourself: Working with Operators

Create a new script including two variables that are assigned numeric values. Add a third variable and assign it the product of the first two variables. Display the value of the third variable on the console.

```
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> {
    var $first-variable = 100;
    var $second-variable = 5;
    var $third-variable = $first-variable * $second-variable;
    <output> "Result: " _ $third-variable;
  }
}
```

Chapter 3: Try It Yourself: Working with Command-line Arguments

Create a new script with a command-line argument that accepts a number from the user. Include the `$arguments` global variable so that the command-line argument will be included in the CLI help output. Perform a mathematical operation on the command-line argument and output the result to the console. Execute the op script a few times, with a different number provided on the command-line each time to verify that the result changes according to the entered number.

```
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";

var $arguments = {
  <argument> {
    <name> "number";
    <description> "The number to multiply.";
  }
}

param $number;

match / {
  <op-script-results> {
    <output> "Result: " _ $number * 55;
  }
}
```

Chapter 3:
Try It Yourself: Conditionally Assigning Variable Values

Create a new script with a command-line argument which can be set to either + or - signifying the mathematical operation that you wish to perform. Create a variable that is assigned conditionally based on the value of the command-line argument. If the command-line argument is specified as a + then two values should be added together and assigned to the variable. If the command-line argument is specified as a - then subtraction should be performed between the two values and assigned to the variable. Output the result to the console.

```
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";

var $arguments = {
  <argument> {
    <name> "operator";
    <description> "Either + or -";
  }
}

param $operator;

match / {
  <op-script-results> {
    var $first = 31;
    var $second = 14;

    var $conditional = {
      if( $operator == "+" ) {
        expr $first + $second;
      }
      else if( $operator == "-" ) {
        expr $first - $second;
      }
```

```
    }
    <output> $conditional;
  }
}
```

Chapter 3:
Try It Yourself: Working with Named Templates

Create a new script that contains a named template. The template should write a string to the result tree. Redirect this into a variable in the calling template and output the variable value to the console.

```
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> {

    var $redirected = { call output-string(); }

    <output> $redirected;
  }
}

template output-string() {
  expr "Here is the output string";
}
```

Chapter 3:
Try It Yourself: Working with Functions

Create a new script with a variable assigned to the value "Juniper Networks". Output the following to the console on separate lines:

1. The variable value

2. The variable value - right justified in a 20 space field

3. The string length of the variable

4. The substring before the space

5. The string converted entirely into upper-case.

```
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> {

    var $variable = "Juniper Networks";
    <output> $variable;
    <output> jcs:printf( "%20s", $variable );
    <output> string-length( $variable );
    <output> substring-before( $variable, " " );
```

```
    <output> translate( $variable, "abcdefghijklmnopqrstuvwxyz",
"ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
  }
}
```

Chapter 4:
Try It Yourself: Invoking Junos Operational Commands

Following the example of the clear-statistics op script shown in this section, create an op script that reboots the system. (Hint: The XML API Element needed is <request-reboot>).

```
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> {

    var $result = jcs:invoke( "request-reboot" );

  }
}
```

Chapter 4:
Try It Yourself: Retrieving Information from Junos

Create a script similar to the show-admin-status.slax example script above, but instead of the Admin Status report the MTU of a physical interface to the screen. The interface to be displayed should be selected through a command-line argument.

```
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";

var $arguments = {
  <argument> {
    <name> "interface";
    <description> "Show MTU of interface";
  }
}

param $interface;

match / {
  <op-script-results> {

    var $results = jcs:invoke( "get-interface-information" );

    var $mtu = $results/physical-interface[name==$interface]/mtu;

    /* Output the interface mtu to the console */
    <output> "The mtu of " _ $interface _ " is " _ $mtu;
  }
}
```

Chapter 4:
Try It Yourself: Retrieving Information from Junos

Create a script that displays the logical interface MTU of all interfaces within your Junos device.

```
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> {

    var $results = jcs:invoke( "get-interface-information" );

    for-each( $results/physical-interface/logical-interface/address-family/
mtu ) {

      if( . != "Unlimited" ) {
        <output> "The family " _ ../address-family-name _ " MTU for " _ ../../
name _ " is " _ .;
      }
    }
  }
}
```

Chapter 4:
Try It Yourself: Interacting with the User

Modify your script that displays the MTU of a single physical interface. Add a check to see if the command-line argument for the interface has been entered. If it has not then request the information from the user through the `jcs:get-input()` 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";

var $arguments = {
  <argument> {
    <name> "interface";
    <description> "Show MTU of interface";
  }
}

param $interface;

match / {
  <op-script-results> {

    var $interface-value = {
      if( string-length( $interface ) > 0 ) {
        expr $interface;
      }
      else {
        expr jcs:get-input( "Enter interface: " );
      }
```

```
    }

    var $results = jcs:invoke( "get-interface-information" );

    var $mtu = $results/physical-interface[name==$interface-value]/mtu;

    /* Output the interface mtu to the console */
    <output> "The mtu of " _ $interface-value _ " is " _ $mtu;
  }
}
```

Chapter 4:
Try It Yourself: Writing to the Syslog

Create an op script that logs the user name, script, product, and hostname to the syslog from the user facility with a severity level of info.

```
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> {

    var $syslog-message = "User: " _ $user _ " Script: " _ $script _ "
Product: " _
              $product _ " Hostname: " _ $hostname;
    expr jcs:syslog( "user.info", $syslog-message );
  }
}
```

Chapter 4:
Try It Yourself: Reading the Junos Configuration

Create an op script that reads the configuration and outputs all the syslog file names to the console.

```
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> {

    var $configuration = jcs:invoke( "get-configuration" );

    for-each( $configuration/system/syslog/file ) {
      <output> "Syslog File: " _ name;
    }
  }
}
```

Appendix B: Supplemental Junos Automation Information from Part Two

This Appendix supplements the information in Part Two by providing three additional event scripts, as well as example solutions to the *Try It Yourself* sections.

Event Script Examples

This first section of the Appendix provides three additional scripts, which highlight the possibilities that event scripts offer and make use of the lessons learned in this volume. Extensive comments are included within the scripts to provide documentation on their structure.

Save <event-script-input>

When designing an event script that makes use of the `<event-script-input>` source tree data, it is useful to have a copy of the expected XML data structure that the event script would receive. The first example is a very simple event script to gather and save this data for later reference. For the desired event policy, this event script is the event policy action with a configured `destination` and `output-filename` and the `output-format` set to `xml`. Then, when the event occurs, Junos executes the script and saves the entire `<event-script-input>` contents in the output file within the `<event-script-results>` parent result tree element.

The minimum version required is Junos 9.3 because of the use of the `<event-script-input>` source tree element.

Here is an example of the expected output:

```
<event-script-results>
<event-script-input>
<trigger-event>
<id>UI_COMMIT</id>
<type>syslog</type>
<generation-time junos:seconds="1251137835">
2009-08-24 11:17:15 PDT
</generation-time>
<process>
<name>mgd</name>
<pid>4192</pid>
</process>
<hostname>Junos</hostname>
<message>UI_COMMIT: User 'user' requested 'commit' operation (comment:
none)</message>
<facility>interact</facility>
<severity>notice</severity>
<attribute-list>
<attribute>
<name>username</name>
<value>user</value>
</attribute>
<attribute>
<name>command</name>
<value>commit</value>
</attribute>
<attribute>
<name>message</name>
<value>none</value>
```

```
</attribute>
</attribute-list>
</trigger-event>
</event-script-input></event-script-results>
```

To achieve the above output, the following event configuration was used:

```
event-options {
   policy on-commit {
      events ui_commit;
      then {
         event-script save-event-script-input.slax {
            output-filename event-script-input;
            destination local;
            output-format xml;
         }
      }
   }
   event-script {
      file save-event-script-input.slax;
   }
   destinations {
      local {
         archive-sites {
            /var/tmp;
         }
      }
   }
}
```

Here is the code for `save-event-script-input.slax`:

```
/*
 * This simple script is used to capture the <event-script-input> received
by
 * an event script and write it in XML format to the output file. It is
intended
 * to get a glimpse into what data an event script will receive from a
particular
 * event policy. For example, if you want to check what the <event-script-
input>
 * would look like for a policy like this:
 *
 * policy on-commit {
 *    events ui_commit;
 *    ....
 * }
 *
 * Then you could configure it like this:
 *
 *
 * policy on-commit {
 *    events ui_commit;
 *    then {
 *       event-script save-event-script-input.slax {
 *          destination local;
 *          output-format xml;
 *          output-filename event-script-input;
 *       }
 *    }
 * }
 *
```

```
 * Minimum JUNOS version is 9.3 because of the use of <event-script-input>
 *
 */

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 / {
    <event-script-results> {

        /*
         * This copies the entire XML contents of <event-script-input> into
         * the result tree as a child element of <event-script-results>.
         */
        copy-of event-script-input;
    }
}
```

Change Route Advertisement

This event script alters an export routing policy-statement based on the number of operational member links of an aggregate-ethernet uplink. The script is designed to work in a specific scenario but can be modified easily to fit other interface names or policy details. Here are the specific configuration settings that the event script is designed to work with:

- ■ Aggregate member links are ge-4/1/0 and ge-5/1/0.
- ■ Policy-statement is "export", term is "advertise".
- ■ If both member links are operational then the route is accepted and local-preference is set to 100.
- ■ If one member link is operational then the route is accepted and local-preference is set to 50.
- ■ If no member links are operational then the route is rejected.

The minimum version required for this event script is Junos 9.3 because configuration changes are being performed with the `jcs:open()` and `jcs:close()` functions and `jcs:load-configuration` template.

The event policy is embedded within the event script so the only configuration needed is the statement to enable the event script:

```
event-options {
    event-script {
        file change-route-advertisement.slax;
    }
}
```

NOTE The `change-route-advertisement.slax` event script assumes that the advertise term is configured correctly at the time the script is enabled. The event script does not check the configuration until one of the member links goes up or down.

Here is the code for the `change-route-advertisement.slax` event script:

```
/*
 * This script is designed to alter the local preference of an export policy
 * based on how many of the ge-links of the aggregate ethernet uplink are
 * functional.
 *
 * The script works with the following configuration:
 *
 * Aggregate Ethernet link with 2 members: ge-4/1/0, ge-5/1/0
 *
 * IBGP export policy named: export with term name: advertise
 *
 * If both links are up then the local-preference is set to 100. If one link
 * is up then the local-preference is set to 50, if neither link is up then
 * the term is changed from accept to reject.
 *
 * Minimum JUNOS version is 9.3 because of the use of jcs:open(),
 * jcs:load-configuration, and jcs:close().
 *
 */

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";

/*
 * This is the embedded event policy. The event script will be executed
anytime
 * either ge-4/1/0 or ge-5/1/0 goes up or down. This is accomplished by
matching
 * on the SNMP_TRAP_LINK_UP and SNMP_TRAP_LINK_DOWN events.
 * These events are matched only for the two interfaces by using attributes-
match
 * condition statements.
 */
var $event-definition = {
   <event-options> {
      <policy> {
         <name> "change-route-advertisement";
         <events> "snmp_trap_link_up";
         <events> "snmp_trap_link_down";
         <attributes-match> {
            <from-event-attribute> "snmp_trap_link_up.interface-name";
            <condition> "matches";
            <to-event-attribute-value> "(ge-4/1/0.0)|(ge-5/1/0.0)";
         }
         <attributes-match> {
            <from-event-attribute> "snmp_trap_link_down.interface-name";
            <condition> "matches";
            <to-event-attribute-value> "(ge-4/1/0.0)|(ge-5/1/0.0)";
         }
         <then> {
            <event-script> {
               <name> "change-route-advertisement.slax";
            }
         }
      }
   }
}
```

```
/*
 * Note that this script does not include the <event-script-results> element.
 * While part of the boilerplate it is not actually required unless the result
 * tree will be used by the script. Because this script does not use any
result
 * tree elements the <event-script-results> element is not necessary. (But it
 * would not cause any problems to include it).
 */
match / {

   /*
    * Gather the interface information, this will tell us if they are up or
    * down.
    */
    var $interface-rpc = {
       <get-interface-information> {
          <terse>;
       }
   }
   var $interfaces = jcs:invoke( $interface-rpc );

   /*
    * The only interfaces we care about are ge-4/1/0 and ge-5/1/0, count how
    * many of them are up.
    */
   var $up-count = count( $interfaces/physical-interface[name == "ge-4/1/0"
||
                name == "ge-5/1/0" ]/logical-interface[oper-status ==
"up"]);

   /*
    * No changes should be made if the policy is already correct. Gather the
    * current policy configuration for comparison.
    */
   var $configuration-rpc = {
      <get-configuration database="committed"> {
         <configuration> {
            <policy-options> {
               <policy-statement> {
                  <name> "export";
               }
            }
         }
      }
   }
   var $current = jcs:invoke( $configuration-rpc );

   /* Get the "export policy term advertise then" element node to simplify
location paths */
   var $then = $current/policy-options/policy-statement[name=="export"]/
term[name=="advertise"]/then;

   /*
    * This next section checks if any changes are needed and makes them if
    * warranted. The results are stored into a variable called $results so
    * that any commit errors can be reported.
    */
   var $results := {

      /* If no interfaces are up then the term should be set to reject. */
      if( $up-count == 0 ) {

         /* Compare against the current policy, if set to accept then we need
```

```
to change it */
          if( $then/accept ) {
             var $configuration = {
                <configuration> {
                   <policy-options> {
                      <policy-statement> {
                         <name> "export";
                         <term> {
                            <name> "advertise";
                            <then replace="replace"> {
                               <reject>;
                            }
                         }
                      }
                   }
                }

             /* Log a syslog message that the script is making a
configuration change. */
             expr jcs:syslog( "external.notice", "Changing route
advertisement to reject." );

             /*
              * Open a connection, load and commit the configuration, and
close the connection.
              * All results are written to the result tree and will be set as
the value of the
              * $results variable.
              */
             var $connection = jcs:open();
             call jcs:load-configuration( $connection, $configuration,
$action = "replace" );
             copy-of jcs:close( $connection );
          }
       }
       /* If one interface is up then the term should be set to accept with a
local-preference of 50 */
       else if( $up-count == 1 ) {

          /* Compare against current policy to see if a change is needed  */
          if( $then/reject || not( $then/local-preference[local-
preference=="50"]  ) ) {
             var $configuration = {
                <configuration> {
                   <policy-options> {
                      <policy-statement> {
                         <name> "export";
                         <term> {
                            <name> "advertise";
                            <then replace="replace"> {
                               <accept>;
                               <local-preference> {
                                  <local-preference> "50";
                               }
                            }
                         }
                      }
                   }
                }

             /* Log a syslog message that the script is making a
```

```
configuration change. */
            expr jcs:syslog( "external.notice", "Changing route
advertisement to local-pref 50." );

            /*
             * Open a connection, load and commit the configuration, and
close the connection.
             * All results are written to the result tree and will be set as
the value of the
             * $results variable.
             */
            var $connection = jcs:open();
            call jcs:load-configuration( $connection, $configuration,
$action = "replace" );
            copy-of jcs:close( $connection );
         }
      }
      /* If both interfaces are up set policy to accept with a local-
preference of 100 */
      else {

         /* Compare against current policy to see if a change is needed  */
         if( $then/reject || not( $then/local-preference[local-
preference=="100"] ) ) {
            var $configuration = {
               <configuration> {
                  <policy-options> {
                     <policy-statement> {
                        <name> "export";
                        <term> {
                           <name> "advertise";
                           <then replace="replace"> {
                              <accept>;
                              <local-preference> {
                                 <local-preference> "100";
                              }
                           }
                        }
                     }
                  }
               }
            }

            /* Log a syslog message that the script is making a
configuration change. */
            expr jcs:syslog( "external.notice", "Changing route
advertisement to local-pref 100." );

            /*
             * Open a connection, load and commit the configuration, and
close the connection.
             * All results are written to the result tree and will be set as
the value of the
             * $results variable.
             */
            var $connection = jcs:open();
            call jcs:load-configuration( $connection, $configuration,
$action = "replace" );
            copy-of jcs:close( $connection );
         }
      }
   }
```

```
    /*
     * Check for any xnm:error results from the commit operations. If any
occur in the $results
     * variable then log them to the syslog
     */
    if( $results//xnm:error ) {
        for-each( $results//xnm:error ) {
            expr jcs:syslog( "external.error", "Error committing
advertisement changes: ", message );
        }
    }
}
```

Static Route Next-Hop Watcher

This event script example shows how to modify the Junos configuration based on the results of Real-Time Performance Monitoring (RPM) tests. The `watch-next-hop.slax` event script activates/deactivates a static route based on the success/failure of the RPM test to the route's next-hop. The event script code is designed to work with the following configuration settings, but could be modified to work with different static routes or RPM tests:

- Static route is 192.168.1.0/24
- RPM probe name is: Next-Hop
- RPM test name is: 10.0.0.1

The minimum version required is Junos 9.4 because the event script uses a system bootup event as a correlating event, which was introduced in that version.

The event policy is embedded within the event script, so this is the only configuration necessary to implement the script:

```
event-options {
    event-script {
        file watch-next-hop.slax;
    }
}
```

NOTE The `watch-next-hop.slax event script` assumes that the static route is properly activated or deactivated at the time the script is enabled. The event script does not check the configuration until the RPM test results change or the system is rebooted.

Here is the code for the `watch-next-hop.slax` event script:

```
/*
 * This event script activates/deactivates a static route based on the
success
 * or failure of a RPM test to the route's next hop. When the test is
successful
 * the route will be activated. When the test fails the route will be
deactivated.
 *
 * This script is hardcoded to work with a single static route and RPM test.
These
 * could be easily modified to meet any unique requirements:
 *
 * Static Route: 192.168.1.0/24
```

```
 * RPM Probe: Next-Hop
 * RPM Test: 10.0.0.1
 *
 * Minimum version is JUNOS 9.4 because of the use of the SYSTEM bootup
 * event "Starting of initial processes complete"
 *
 */

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";

/*
 * The embedded event policy - the script should be executed when a
 * PING_TEST_FAILED or PING_TEST_COMPLETED event occur with the indicated
 * test-owner and test-name attributes for the event.
 *
 * within conditions are included in each policy to ensure that the script
 * will not be repeatedly called when no change is occurring. The policy
 * will only execute the script if the device recently booted or if the RPM
 * probe result recently changed from failed to completed or vice versa.
 */
var $event-definition = {
   <event-options> {
      <policy> {
         <name> "next-hop-probe-failed";
         <events> "ping_test_failed";
         <attributes-match> {
            <from-event-attribute> "ping_test_failed.test-owner";
            <condition> "matches";
            <to-event-attribute-value> "Next-Hop";
         }
         <attributes-match> {
            <from-event-attribute> "ping_test_failed.test-name";
            <condition> "matches";
            <to-event-attribute-value> "10.0.0.1";
         }
         <attributes-match> {
            <from-event-attribute> "system.message";
            <condition> "matches";
            <to-event-attribute-value> "Starting of initial processes
complete";
         }
         <within> {
            <name> "600";
            <events> "ping_test_completed";
            <events> "system";
         }
         <then> {
            <event-script> {
               <name> "watch-next-hop.slax";
            }
         }
      }
      <policy> {
         <name> "next-hop-probe-succeeded";
         <events> "ping_test_completed";
         <attributes-match> {
            <from-event-attribute> "ping_test_completed.test-owner";
```

```
            <condition> "matches";
            <to-event-attribute-value> "Next-Hop";
         }
         <attributes-match> {
            <from-event-attribute> "ping_test_completed.test-name";
            <condition> "matches";
            <to-event-attribute-value> "10.0.0.1";
         }
         <attributes-match> {
            <from-event-attribute> "system.message";
            <condition> "matches";
            <to-event-attribute-value> "Starting of initial processes
complete";
         }
         <within> {
            <name> "600";
            <events> "ping_test_failed";
            <events> "system";
         }
         <then> {
            <event-script> {
               <name> "watch-next-hop.slax";
            }
         }
      }
   }
}

match / {
   <event-script-results> {
      /* Learn the event type, either a PING_TEST_FAILED or PING_TEST_
COMPLETED */
      var $event-type = event-script-input/trigger-event/id;

      /* Retrieve the current configuration for the static route */
      var $configuration-rpc = {
         <get-configuration database="committed"> {
            <configuration> {
               <routing-options>;
            }
         }
      }
      var $current = jcs:invoke( $configuration-rpc );

      /* Grab the routing-options static node to make further location
paths shorter */
      var $static = $current/routing-options/static;

      /* Is the route currently inactive? */
      var $inactive = $static/route[name == "192.168.1.0/24"]/@inactive;

      /*
       * Compare the event type vs the current value of $inactive. If they
       * do not match then a configuration change must be performed.
       */

      /* RPM test failed but the route is currently active */
      if( $event-type == "PING_TEST_FAILED" && jcs:empty( $inactive ) ) {

         /* Needed configuration change */
         var $configuration = {
            <configuration> {
               <routing-options> {
```

```
                    <static> {
                        <route inactive="inactive"> {
                            <name> "192.168.1.0/24";
                        }
                    }
                }
            }
        }

        /* Open connection, load and commit the change, and close
connection  */
        var $connection = jcs:open();
        var $results := {
            call jcs:load-configuration( $connection, $configuration );
            copy-of jcs:close( $connection );
        }

        /* If any errors occurred during the commit process then report them
to the syslog */
        if( $results//xnm:error ) {
            for-each( $results//xnm:error ) {
                expr jcs:syslog( "external.error", "Error deactivating
192.168.1.0/24: ", message );
            }
        }
        /* Otherwise, report success */
        else {
            expr jcs:syslog( "external.notice", "Static route
192.168.1.0/24 disabled." );
        }
    }
    /* RPM test succeeded but the route is currently inactive */
    else if( $event-type == "PING_TEST_COMPLETED" && $inactive ) {

        /* Needed configuration change */
        var $configuration = {
            <configuration> {
                <routing-options> {
                    <static> {
                        <route active="active"> {
                            <name> "192.168.1.0/24";
                        }
                    }
                }
            }
        }

        /* Open connection, load and commit the change, and close
connection  */
        var $connection = jcs:open();
        var $results := {
            call jcs:load-configuration( $connection, $configuration );
            copy-of jcs:close( $connection );
        }

        /* If any errors occurred during the commit process then report them
to the syslog */
        if( $results//xnm:error ) {
            for-each( $results//xnm:error ) {
                expr jcs:syslog( "external.error", "Error activating
192.168.1.0/24: ", message );
            }
        }
```

```
            /* Otherwise, report success */
            else {
               expr jcs:syslog( "external.notice", "Static route
192.168.1.0/24 activated." );
            }
         }
      }
}
```

Try It Yourself Solutions in Part Two

This section of the Appendix provides sample solutions for each of the *Try It Yourself* sections in Chapters 5 through 8.

Chapter 5
Try It Yourself: Simulating Events with the Logger Utility

1. Use `help syslog` to identify an event of interest and its attributes.

2. Configure a syslog file to use structured-data format.

3. Using the logger utility, generate an artificial version of the selected event including values for all of its attributes.

4. Verify that the event was created as expected by viewing the structured-data syslog file.

```
user@Junos> help syslog RPD_IGMP_JOIN
Name:        RPD_IGMP_JOIN
Message:     Listener <source-address> sent a join to <destination-
address> for group <group-address> source
            <sender-address> on interface <interface-name> at <time>
Help:        IGMP join event
Description:  IGMP join event.
Type:        Event: This message reports an event, not an error
Severity:    info
```

```
system {
   syslog {
      file structured {
         any any;
         structured-data;
      }
   }
}
```

```
user@Junos> start shell
% logger -e RPD_IGMP_JOIN -a source-address=10.0.0.1 -a destination-
address=10.0.0.2 -a group=225.1.1.1
```

```
user@Junos> show log structured | match RPD_IGMP_JOIN
<13>1 2009-08-27T07:22:49.394-07:00 JUNOS logger - RPD_IGMP_JOIN
[junos@2636.1.1.1.2.9 source-address="10.0.0.1" destination-
address="10.0.0.2" group="225.1.1.1"]
```

Chapter 5
Try It Yourself: Logging a Syslog Message in Response to an Event

1. Using the `log-hello-world.slax` script as an example, create an event script that logs a message to the syslog.

2. Copy the event script to the Junos device and enable the script in the configuration.

3. Select an event ID of interest and configure an event policy that executes the event script in response to the system event.

4. Use the logger utility to simulate the event and verify that the desired message is logged to the syslog.

```
/* log-syslog.slax */
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 / {
   <event-script-results> {

      /* Send to the syslog */
      expr jcs:syslog("external.info", "I‘ve been executed!");

   }
}
```

```
user@Junos> file list /var/db/scripts/event/log-syslog.slax
/var/db/scripts/event/log-syslog.slax
```

```
event-options {
   policy log-message {
      events rpd_igmp_join;
      then {
         event-script log-syslog.slax;
      }
   }
   event-script {
      file log-syslog.slax;
   }
}
```

```
user@Junos> start shell
% logger -e RPD_IGMP_JOIN
```

```
user@Junos> show log messages | match cscript
Aug 27 07:30:01  JUNOS cscript: I‘ve been executed!
```

Try It Yourself: Matching Nonstandard Events

Find a nonstandard event that has been logged to the syslog of your Junos device. Craft an event policy that matches this event and executes an event script. The event script should write a message to the syslog indicating that the script was executed.

```
user@Junos> show log messages | match "becoming master"
Aug 27 08:07:46  JUNOS /kernel: mastership: routing engine 0 becoming master
```

```
event-options {
   policy log-message {
      events kernel;
      attributes-match {
         kernel.message matches "routing engine.*becoming master";
      }
      then {
         event-script log-syslog.slax;
      }
   }
   event-script {
      file log-syslog.slax;
   }
}
```

```
/* log-syslog.slax */
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 / {
   <event-script-results> {

      /* Send to the syslog */
      expr jcs:syslog("external.info", "I've been executed!");

   }
}
```

Chapter 5
Try It Yourself: Time-based Configuration Changes

Create a local user account called "test-user" on your Junos device. Create the necessary generated events, event policies, and event scripts to have the "test-user" automatically assigned to the super-user class from 8am to 5pm and the read-only class from 5pm to 8am.

```
event-options {
   event-script {
      file change-user-class.slax;
   }
}
```

```
/* change-user-class.slax */
```

```
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";

/* Embedded event policy */
var $event-definition = {
    <event-options> {
        <generate-event> {
            <name> "08:00";
            <time-of-day> "08:00:00 +0000";
        }
        <generate-event> {
            <name> "17:00";
            <time-of-day> "17:00:00 +0000";
        }
        <policy> {
            <name> "change-user-class";
            <events> "08:00";
            <events> "17:00";
            <then> {
                <event-script> {
                    <name> "change-user-class.slax";
                }
            }
        }
    }
}

match / {
    <event-script-results> {

        /* Determine which event triggered the script */
        var $event-id = event-script-input/trigger-event/id;

        if( $event-id == "08:00" ) {
            /* Change user to super-user */
            var $configuration = {
                <configuration> {
                    <system> {
                        <login> {
                            <user> {
                                <name> "test-user";
                                <class> "super-user";
                            }
                        }
                    }
                }
            }
            /* Open connection, load and commit, close connection */
            var $connection = jcs:open();
            var $results := {
                call jcs:load-configuration( $connection, $configuration );
                copy-of jcs:close( $connection );
            }

            /* If any errors occurred during the commit process then report them
to the syslog */
            if( $results//xnm:error ) {
                for-each( $results//xnm:error ) {
```

```
                expr jcs:syslog( "external.error", "Error setting test-user
to super-user", message );
            }
         }
         /* Otherwise, report success */
         else {
            expr jcs:syslog( "external.notice", "test-user set to super-
user" );
         }
      }
      else {
         /* Change user to read-only */
         var $configuration = {
            <configuration> {
               <system> {
                  <login> {
                     <user> {
                        <name> "test-user";
                        <class> "read-only";
                     }
                  }
               }
            }
         }
         /* Open connection, load and commit, close connection */
         var $connection = jcs:open();
         var $results := {
            call jcs:load-configuration( $connection, $configuration );
            copy-of jcs:close( $connection );
         }

         /* If any errors occurred during the commit process then report
them to the syslog */
         if( $results//xnm:error ) {
            for-each( $results//xnm:error ) {
               expr jcs:syslog( "external.error", "Error setting test-user
to read-only", message );
            }
         }
         /* Otherwise, report success */
         else {
            expr jcs:syslog( "external.notice", "test-user set to read-
only" );
         }
      }

   }
}
```

Chapter 6
Try It Yourself: Executing Commands

Create an event policy that reacts to a `UI_COMMIT` event by storing the configuration of the user account that performed the commit. The output file should be saved locally in the /var/tmp directory in XML format.

```
event-options {
   policy save-user-config {
      events ui_commit;
      then {
         execute-commands {
            commands {
```

```
                    "show configuration system login user {$$.username}";
                }
                output-filename user-config;
                destination local;
                output-format xml;
            }
        }
    }
    destinations {
        local {
            archive-sites {
                /var/tmp;
            }
        }
    }
}
```

Chapter 6
Try It Yourself: Uploading Files

Create an event policy that uploads the `messages` log file to a remote server every day.

```
event-options {
    generate-event {
        midnight time-of-day "00:00:00 +0000";
    }
    policy upload-messages {
        events midnight;
        then {
            upload filename /var/log/messages destination remote;
        }
    }
    destinations {
        remote {
            archive-sites {
                "ftp://user@10.0.0.1" password "$9$nWgP6tO1IclvLEcVw2gJZ69C";
## SECRET-DATA
            }
        }
    }
}
```

Chapter 6
Try It Yourself: Raising SNMP Traps

Create an event policy that raises an SNMP trap every time a user enters or leaves the configuration database.

```
event-options {
    policy raise-trap {
        events [ ui_dbase_login_event ui_dbase_logout_event ];
        then {
            raise-trap;
        }
    }
}
```

Chapter 6
Try It Yourself: Ignoring Events

Add an ignore-event policy before the event policy that was created in the Executing Commands section. This ignore-event policy should run if a commit is performed more than once per minute.

```
event-options {
    policy ignore-commit {
        events ui_commit;
        within 60 {
            trigger after 1;
        }
        then {
            ignore;
        }
    }
    policy save-user-config {
        events ui_commit;
        then {
            execute-commands {
                commands {
                    "show configuration system login user {$$.username}";
                }
                output-filename user-config;
                destination local;
                output-format xml;
            }
        }
    }
    destinations {
        local {
            archive-sites {
                /var/tmp;
            }
        }
    }
}
```

Chapter 8
Try It Yourself: Logging Out Users

Using the `clear bgp neighbor` command without specifying a peer address causes all BGP peers to be reset. Write an event policy and event script that automatically disconnects any user who runs this command without including a peer address.

```
event-options {
    policy logout-user {
        events ui_cmdline_read_line;
        attributes-match {
            ui_cmdline_read_line.command matches "^(run )?clear bgp neighbor
$";
        }
        then {
            event-script user-logout.slax {
                arguments {
                    username "{$$.username}";
                }
            }
        }
    }
    event-script {
        file user-logout.slax;
    }
}


/* user-logout.slax */
```

```
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";

/* Username argument */
param $username;

match / {

   /* Logout the user */
   var $command = <command> "request system logout user " _ $username;

   var $results = jcs:invoke( $command );

   /* If any errors occurred then report them to the syslog */
   if( $results/..//xnm:error ) {
      for-each( $results/..//xnm:error ) {
         expr jcs:syslog( "external.error", "Error logging out ", $username,
": ", message);
      }
   }
   else {
      var $message = "Logged out " _ $username _ " for clearing all BGP
neighbors.";
      expr jcs:syslog( "external.notice", $message );
   }
}
```

Chapter 8
Try It Yourself: Dampening Event Reactions

Chapter 3 included a save-core event policy that demonstrated the `upload filename` policy action. Using that event policy as a guideline, create a policy that executes an event script in response to an `eventd` core dump. The event script should upload all eventd core files to a remote server, but the action should be dampened by the `jcs:dampen()` function to a maximum of 1 time per minute. When this limit is exceeded a syslog message should be logged instead, indicating that the core upload process was dampened.

```
event-options {
   event-script {
      file save-cores.slax;
   }
}


/* save-cores.slax */
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";

/* Embedded event policy */
var $event-definition = {
   <event-options> {
      <policy> {
         <name> "save-core";
```

```
            <events> "kernel";
            <attributes-match> {
               <from-event-attribute> "kernel.message";
               <condition> "matches";
               <to-event-attribute-value> "(eventd).*(core dumped)";
            }
            <then> {
               <event-script> {
                  <name> "save-cores.slax";
               }
            }
         }
      }
}

match / {
   <event-script-results> {

      if( jcs:dampen( "save-core", 1, 1 ) ) {
         var $rpc = {
            <file-list> {
               <path> "/var/tmp/eventd.core*";
            }
         }
         var $file-list = jcs:invoke( $rpc );

         /* Pause a few seconds to let the core be gathered */
         expr jcs:sleep(5);

         for-each( $file-list/directory/file-information/file-name ) {
            var $copy-rpc = {
               <file-copy> {
                  <source> .;
                  <destination> "ftp://user:password@10.0.0.1";
               }
            }
            var $results = jcs:invoke( $copy-rpc );

            /* If any errors occurred then report them to the syslog */
            if( $results/..//xnm:error ) {
               for-each( $results/..//xnm:error ) {
                  expr jcs:syslog( "external.error", "Error copying eventd
cores: ", message );
               }
            }
         }
      }
      else {
         expr jcs:syslog("external.notice", "Dampening eventd core
upload." );
      }
   }
}
```

Chapter 8
Try It Yourself: Embedding Event Policy

Choose an event policy and event script that you created in one of the prior *Try It Yourself* exercises. Remove the event policy from the configuration and embed the policy within the event script.

```
event-options {
   event-script {
```

```
        file log-syslog-embedded.slax;
    }
}


/* log-syslog-embedded.slax */
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";

var $event-definition = {
    <event-options> {
        <policy> {
            <name> "log-message";
            <events> "rpd_igmp_join";
            <then> {
                <event-script> "log-syslog-embedded.slax";
            }
        }
    }
}

match / {
    <event-script-results> {

        /* Send to the syslog */
        expr jcs:syslog("external.info", "I've been executed!");

    }
}
```

Chapter 8
Try It Yourself: Using <event-script-input>

Revise your earlier event script that automatically logged out users that used the `clear bgp neighbor` command without specifying a peer address. Remove the event policy from the configuration and embed the policy within the event script's `$event-definition` variable. Remove any command-line arguments used to communicate which user performed the command and instead use the `<event-script-input>` source tree element to determine which user should be logged out.

```
event-options {
    event-script {
        file logout-user.slax;
    }
}


/* logout-user.slax */
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";
```

```
var $event-definition = {
   <event-options> {
      <policy> {
         <name> "logout-user";
         <events> "ui_cmdline_read_line";
         <attributes-match> {
            <from-event-attribute> "ui_cmdline_read_line.command";
            <condition> "matches";
            <to-event-attribute-value> "^(run )?clear bgp neighbor $";
         }
         <then> {
            <event-script> {
               <name> "logout-user.slax";
            }
         }
      }
   }
}

match / {

   /* Get the user name */
   var $username = event-script-input/trigger-event/attribute-list/
attribute[name=="username"]/value;

   /* Logout the user */
   var $command = <command> "request system logout user " _ $username;
 º
   var $results = jcs:invoke( $command );

   /* If any errors occurred then report them to the syslog */
   if( $results/..//xnm:error ) {
      for-each( $results/..//xnm:error ) {
         expr jcs:syslog( "external.error", "Error logging out ",
$username, ": ", message);
      }
   }
   else {
      var $message = "Logged out " _ $username _ " for clearing all BGP
neighbors.";
      expr jcs:syslog( "external.notice", $message );
   }
}
```

Appendix C: Supplemental Junos Automation Information from Part Three

This Appendix supplements the information discussed Part Three by providing an additional commit script example as well as examples of solutions to the *Try It Yourself* sections.

Commit Script Example

This first section of Appendix C provides an additional commit script highlighting the possibilities of commit scripts and making use of the lessons learned in this volume. Extensive comments are included within the script to provide documentation on its structure.

Policy-Based Routing

Junos performs policy-based routing through filter-based forwarding. A firewall filter term directs its matching traffic to an alternate table for forwarding, and this assigned forwarding table is populated with the necessary routes to send the traffic to its desired destination.

The multi-part approach behind filter-based forwarding provides great flexibility, but it can become cumbersome when only a simple next-hop directive is desired.

The policy-route.slax commit script provides a simplified method of policy-based routing. The only action necessary to direct traffic to a remote destination is to apply the following macro under the firewall filter term's `then` hierarchy:

```
apply-macro policy-route {
   next-hop x.x.x.x;
}
```

Here is an example of a firewall filter that uses this macro to direct traffic from two specific sources to different destinations:

```
firewall {
   family inet {
      filter customer-input {
         term source-a {
            from {
               source-address {
                  192.168.1.1/32;
               }
            }
            then {
               apply-macro policy-route {
                  next-hop 10.0.0.1;
               }
            }
         }
         term source-b {
            from {
               source-address {
                  192.168.1.14/32;
               }
            }
            then {
               apply-macro policy-route {
                  next-hop 10.0.0.2;
               }
```

```
            }
        }
    }
}
```

Using this macro significantly reduces the number of configuration statements that an administrator must enter to set up policy-based routing. The commit script adds all other necessary configuration statements during the commit process in response to the presence of the policy-route macros. Junos adds these statements transiently in order to prevent cluttering the configuration, and so the only permanent configuration statements that refer to policy-based routing are the `apply-macro` statements.

NOTE As discussed in Chapter 11, transient changes are communicated to Junos and affect its operation but do not appear in the configuration file.

Here is the configuration that is added based on the above firewall filter and its macros:

```
routing-options {
    interface-routes {
        rib-group inet fbf-ribs;
    }
    rib-groups {
        fbf-ribs {
            import-rib [ inet.0 fbf-10.0.0.1.inet.0 fbf-10.0.0.2.inet.0 ];
        }
    }
}
firewall {
    family inet {
        filter customer-input {
            term source-a {
                then {
                    routing-instance fbf-10.0.0.1;
                }
            }
            term source-b {
                then {
                    routing-instance fbf-10.0.0.2;
                }
            }
        }
    }
}
routing-instances {
    fbf-10.0.0.1 {
        instance-type forwarding;
        routing-options {
            static {
                route 0.0.0.0/0 next-hop 10.0.0.1;
            }
        }
    }
    fbf-10.0.0.2 {
        instance-type forwarding;
        routing-options {
            static {
```

```
            route 0.0.0.0/0 next-hop 10.0.0.2;
        }
    }
  }
}
```

During the commit process Junos merges the above changes into the existing configuration as transient changes. While the changes do not appear in the configuration file, their effects can be seen in the routing tables that are created to accommodate the two new forwarding instances:

```
fbf-10.0.0.2.inet.0: 5 destinations, 5 routes (5 active, 0 holddown, 0
hidden)
+ = Active Route, - = Last Active, * = Both

0.0.0.0/0          *[Static/5] 00:12:05
                   > to 10.0.0.2 via ge-2/0/0.0
10.0.0.0/24        *[Direct/0] 00:12:05
                   > via ge-2/0/0.0
10.0.0.100/32      *[Local/0] 00:12:05
                    Local via ge-2/0/0.0
192.168.1.0/24     *[Direct/0] 00:12:05
                   > via ge-4/1/0.0
192.168.1.50/32    *[Local/0] 00:12:05
                    Local via ge-4/1/0.0

fbf-10.0.0.1.inet.0: 5 destinations, 5 routes (5 active, 0 holddown, 0
hidden)
+ = Active Route, - = Last Active, * = Both

0.0.0.0/0          *[Static/5] 00:12:05
                   > to 10.0.0.1 via ge-2/0/0.0
10.0.0.0/24        *[Direct/0] 00:12:05
                   > via ge-2/0/0.0
10.0.0.100/32      *[Local/0] 00:12:05
                    Local via ge-2/0/0.0
192.168.1.0/24     *[Direct/0] 00:12:05
                   > via ge-4/1/0.0
192.168.1.50/32    *[Local/0] 00:12:05
                    Local via ge-4/1/0.0
```

The policy-route.slax commit script functions correctly with an existing `interface-routes rib-group`, as long as it does not have an `import-policy`. Here is the full list of caveats:

- Logical-systems are not supported.
- Only IPv4 is supported.
- An `import-policy` for the `interface-routes rib-group` is not supported.
- Deactivating the `[edit routing-options]` hierarchy will prevent the script from functioning correctly.
- The next-hop must be to a direct subnet.

Here is the full listing of the policy-route.slax commit script:

```
/*
 * policy-route.slax is a commit script designed to simplify filter-based
 * forwarding (policy-based routing).  With this commit script in place, the
 * following firewall action can be specified:
 *
```

```
 * then {
 *    apply-macro policy-route {
 *       next-hop 10.0.0.1;
 *    }
 * }
 *
 * The commit script translates the configuration macro at commit time and
 * generates the configuration necessary for filter-based forwarding.
 *
 * Do not configure a terminating action or 'next term' within the same term
 * where this macro is applied
 * Next-hop must be to a directly connected interface
 * Only IPv4 is supported
 * Logical-systems are not supported
 * Deactivating routing-options blocks the macro from working correctly
 */
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 {

   /*
    * Verify that a rib-group with an import-rib is not specified for
interface
    * routes.  This commit script is incompatible with that configuration.
    */
   var $interface-routes = routing-options/interface-routes/rib-group/
inet;
   var $rib-group-config = routing-options/rib-groups[name == $interface-
routes];
   if( $interface-routes && $rib-group-config/import-policy ) {
      <xnm:warning> {
         call jcs:edit-path( $dot = $rib-group-config );
         call jcs:statement( $dot = $rib-group-config/import-policy );
         <message> "policy-route macro cannot be used if interface-routes "
_
                  "rib-group has an import policy";
      }
   }
   /* Otherwise, the configuration is compatible so go ahead */
   else {

      /*
       * Go through all the firewall filters and create the transient filter
       * configuration change as well as the routing-instance change.  Build
       * a set of all the routing-instances for the later interface-routes
       * change.
       */
       var $macro = "policy-route";
       var $results := {
          for-each( firewall//filter/term/then/apply-macro[name == $macro]
) {

            /* Retrieve next-hop value from the macro */
            var $next-hop = data[ name == "next-hop" ]/value;

            /* Verify that next-hop is present */
            if( jcs:empty( $next-hop ) ) {
```

```
                    <xnm:warning> {
                        call jcs:edit-path();
                        <message> "Macro is missing next-hop parameter";
                    }
                }
                /* Make changes */
                else {
                    /* Assemble standardized name */
                    var $instance-name = "fbf-" _ $next-hop;

                    /* Add routing-instance action */
                    var $content = {
                        <routing-instance> {
                            <routing-instance-name> $instance-name;
                        }
                    }
                    call jcs:emit-change($dot= ..,$content,$tag="transient-
change");

                    /*
                     * Create routing-instance.  It is a forwarding type instance
                     * with a single 0/0 route pointing to the desired next-hop
                     */
                    <transient-change> {
                        <routing-instances> {
                            <instance> {
                                <name> $instance-name;
                                <instance-type> "forwarding";
                                <routing-options> {
                                    <static> {
                                        <route> {
                                            <name> "0.0.0.0/0";
                                            <next-hop> $next-hop;
                                        }
                                    }
                                }
                            }
                        }
                    }

                    /* Record routing-instance name */
                    <instance> $instance-name;
                }
            }
        }

        /*
         * Copy any <transient-change> elements saved to $results to the
result
         * tree so the changes can be passed to Junos
         */
        copy-of $results/transient-change;
        /*
         * Copy any <xnm:warning> elements saved to $results as well
         */
        copy-of $results/xnm:warning;

        /*
         * Make routing-options change.  The active="active" tag is included
         * up until the routing-options hierarchy in case the interface-routes
         * statement or its children are deactivated.  The macro could have
         * activated routing-options automatically as well, but it does not
         * due to the possibility that there might be configuration within
```

```
         * routing-options that must remain deactivated.
         */
        if( count( $results/instance ) > 0 ) {
           <transient-change> {
              <routing-options> {
                 <interface-routes active="active"> {
                    <rib-group active="active"> {
                       <inet active="active"> "fbf-ribs";
                    }
                 }
                 <rib-groups> {
                    <name> "fbf-ribs";
                    /* Is there an existing interface-routes rib? */
                    if( $interface-routes ) {
                       /* Copy existing ribs to import-rib */
                       copy-of $rib-group-config/import-rib;
                    }
                    else {
                       /* Just add inet.0 as import rib */
                       <import-rib> "inet.0";
                    }

                    /* Add all the routing-instances as import-ribs */
                    for-each( $results/instance ) {
                       <import-rib> . _ ".inet.0";
                    }
                 }
              }
           }
        }
     }
}
```

Try It Yourself Solutions

This last section of the Appendix provides sample solutions for each of the *Try It Yourself* sections as they appeared in Chapters 10 through 12.

Chapter 10
Try It Yourself: Host-Name Should Inherit From Configuration Group

Create a commit script that generates a commit warning message if the host-name is not inherited from the re0 or re1 configuration groups.

```
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( jcs:empty( system/host-name[@junos:group == "re0" || @junos:group ==
"re1"])){
      <xnm:warning> {
         <message> "Hostname is not inherited from re configuration
group.";
      }
```

```
    }

}
```

Chapter 10
Try It Yourself: ISIS Interface Lacks Family Iso

Create a warning message for every interface enabled for the ISIS protocol that does not have family iso configured. Include an <edit-path> to better document the problem.

```
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 {

    /* Record reference point */
    var $interfaces = interfaces;

    /* Only look for specifically enabled interfaces */
    for-each( protocols/isis/interface[ name != "all" ][ jcs:empty( disable
)] ) {
        var $physical = substring-before( name, "." );
        var $logical = substring-after( name, "." );

        var $interface = $interfaces/interface[name == $physical]/unit[name ==
$logical];

        if( jcs:empty( $interface/family/iso ) ) {
            <xnm:warning> {
                call jcs:edit-path();
                <message> "Interface does not have family iso configured.";
            }
        }
    }

}
```

Chapter 10
Try It Yourself: Compare Syslog Methods

Create a commit script that logs two syslog messages, one using <syslog> and the other using jcs:syslog(). Compare the syslog results when a commit is performed versus a commit check.

```
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 {

    <syslog> {
        <message> "Logged by result tree element";
    }
```

```
    expr jcs:syslog( "daemon.warning", "Logged by function" );

}

[edit]
jnpr@host1# run clear log syslog

[edit]
jnpr@host1# commit
commit complete
[edit]
jnpr@host1# run show log syslog | match cscript
Nov 30 09:22:26  host1 cscript: %DAEMON-4: Logged by function
Nov 30 09:22:26  host1 cscript: %DAEMON-4: Logged by result tree element
Nov 30 09:22:37  host1 mgd[1913]: %INTERACT-6-UI_CMDLINE_READ_LINE: User
'jnpr', command 'run show log syslog | match cscript '

[edit]
jnpr@host1# run clear log syslog

[edit]
jnpr@host1# commit check
configuration check succeeds

[edit]
jnpr@host1# run show log syslog | match cscript
Nov 30 09:22:46  host1 cscript: %DAEMON-4: Logged by function
Nov 30 09:22:58  host1 mgd[1913]: %INTERACT-6-UI_CMDLINE_READ_LINE: User
'jnpr', command 'run show log syslog | match cscript '
```

Chapter 10
Try It Yourself: Sanity Checking

Write a commit script that generates a `<xnm:error>` if the `[edit system]`, `[edit interfaces]`, or `[edit protocols]` hierarchies are missing.

```
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( jcs:empty( system ) ) {
        <xnm:error> {
            <message> "[edit system] hierarchy level is missing.";
        }
    }

    if( jcs:empty( interfaces ) ) {
        <xnm:error> {
            <message> "[edit interfaces] hierarchy level is missing.";
        }
    }

    if( jcs:empty( protocols ) ) {
        <xnm:error> {
            <message> "[edit protocols] hierarchy level is missing.";
```

```
        }
    }
}
```

Chapter 10
Try It Yourself: Incorrect Autonomous-System Number

Write a commit script that generates a <xnm:error> if the autonomous-system number is not set to 65000. Include <edit-path> and <statement> elements to better document the problem.

```
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( routing-options/autonomous-system/as-number != 65000 ) {
        <xnm:error> {
            call jcs:edit-path( $dot = routing-options/autonomous-system );
            call jcs:statement( $dot = routing-options/autonomous-system/
as-number );
            <message> "ASN must be set to 65000.";
        }
    }
}
```

Chapter 11
Try It Yourself: Commit Check And The <Change> Element

Write a simple commit script that changes a single configuration setting. Perform a `commit check` and verify that the candidate configuration is altered but the committed configuration remains unchanged. Perform a normal `commit` and verify that the change is now visible in the committed configuration.

```
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 {

    <change> {
        <snmp> {
            <location> "SLC";
        }
    }
}


[edit]
jnpr@host1# show snmp
```

```
location Denver;

[edit]
jnpr@host1# commit check
configuration check succeeds

[edit]
jnpr@host1# show snmp
location SLC;

[edit]
jnpr@host1# run show configuration snmp
location Denver;

[edit]
jnpr@host1# commit
commit complete

[edit]
jnpr@host1# show snmp
location SLC;

[edit]
jnpr@host1# run show configuration snmp
location SLC;
```

Chapter 11
Try It Yourself: Automated Configuration Fixes

Identify a standard part of your configuration that should always be present. Write a commit script that automatically adds it when missing and generates a <xnm:warning> message informing the user of the change.

```
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( jcs:empty( routing-options/autonomous-system[as-number == 65000 ] )
) {
       <change> {
          <routing-options> {
             <autonomous-system> {
                <as-number> 65000;
             }
          }
       }
       <xnm:warning> {
          <edit-path> "[edit routing-options]";
          <message> "Setting ASN to 65000";
       }
   }
}
```

Chapter 11
Try It Yourself: Replacing Configuration Hierarchies

Create a commit script that enforces the requirement that the ospf configuration should consist solely of an assignment of all interfaces into area 0.

```
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 {

   /* Check if invalid configuration */
   if( jcs:empty( protocols/ospf/area[name == "0.0.0.0"]/interface[name ==
"all"] ) ||
       count( protocols/ospf/descendant::* ) != 4 ) {

       <change> {
          <protocols> {
             <ospf replace="replace"> {
                <area> {
                   <name> "0.0.0.0";
                   <interface> {
                      <name> "all";
                   }
                }
             }
          }
       }
       <xnm:warning> {
          <edit-path> "[edit protocols ospf]";
          <message> "Assigning all interfaces to area 0.0.0.0";
       }
   }
}
```

Chapter 11
Try It Yourself: Family Mpls On LDP Interfaces

Create a commit script that calls the `jcs:emit-change` template to add family mpls to every interface, configured under `[edit protocols ldp]`, that lack it.

```
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 {

   /* Save reference */
   var $interfaces = interfaces;

   for-each( protocols/ldp/interface ) {
```

```
        var $physical = substring-before( name, "." );
        var $logical = substring-after( name, "." );
        var $interface = $interfaces/interface[name == $physical]/unit[name
== $logical];

        if( jcs:empty( $interface/family/mpls ) ) {
            var $content = {
                <family> {
                    <mpls>;
                }
            }
            var $message = "Adding family mpls to interface";
            call jcs:emit-change( $dot = $interface, $content, $message );
        }

    }
}
```

Chapter 11
Try It Yourself: Deleting Invalid Name-Servers

Create a commit script for an organization whose name-servers all fall within the 10.0.1.0/24 subnet. Delete any configured name-servers from outside that subnet.

```
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 {

    /* This script does not work with inherited name-servers */
    for-each( system/name-server ) {
        if( not( starts-with( name, "10.0.1." ) ) ) {
            var $content = {
                <name-server delete="delete"> {
                    <name> name;
                }
            }
            var $message = "Removing invalid name-server";
            call jcs:emit-change( $dot = .., $content, $message );
        }
    }
}
```

Chapter 11
Try It Yourself: Reorder Firewall Terms

Create a commit script that adds a term to a firewall filter, if missing, and then inserts it at the beginning of the filter.

```
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 {

   /* Check if term needs to be added */
   var $filter = firewall/family/inet/filter[name == "ingress"];
   if( jcs:empty( $filter/term[1][name == "count"] ) ) {
      var $content1 = {
         <term> {
            <name> "count";
            <then> {
               <count> "counter";
            }
         }
      }
      var $term1-name = $filter/term[1]/name;
      var $message = "Adding count term to ingress filter";
      call jcs:emit-change( $dot = $filter, $content = $content1, $message
);
      var $content2 = {
         <term insert="before" name=$term1-name> {
            <name> "count";
         }
      }
      call jcs:emit-change( $dot = $filter, $content = $content2 );
   }
}
```

Chapter 11
Try It Yourself: Modify Convert-To-Hyphens.Slax

Modify the convert-to-hyphens.slax commit script. Along with renaming the prefix-list, the references to the prefix-list in policy-statements and firewall filters should also be set to the new name.

```
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 {

   /* Not designed for logical systems */

   /* Loop through all prefix-lists */
   for-each( policy-options/prefix-list ) {
      call convert();
   }

   /* Loop through all policy-statement - prefix-lists */
   for-each( policy-options/policy-statement//from/prefix-list ) {
      call convert();
   }

   /* Loop through all firewalls - prefix-lists */
   for-each( firewall//filter/term/from/prefix-list ) {
      call convert();
```

```
    }
}

/* Perform conversion at current hierarchy */
template convert() {

    /* Do they have an underscore in their name? */
    if( contains( name, "_" ) ) {

        /* Translate _ to - */
        var $new-name = translate( name, "_", "-" );
        var $content = {
            <prefix-list rename="rename" name=$new-name> {
                <name> name;
            }
        }
        var $message = "Translating _ to -";
        call jcs:emit-change( $dot=.., $content, $message );
    }
}
```

Chapter 11
Try It Yourself: Transient Root Authentication Key

Create a commit script that adds the root authentication key transiently to the configuration. Use the `jcs:emit-change` template to do so.

```
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 {
    var $content = {
        <root-authentication> {
            <ssh-dsa> {
                <name> "ssh-dss AAAAB3NzaC1kc3MAAACBAM5Yu7v/VlAYXzZ5" _
                       "XUDmBwAGgARS4ILMlhU2ozpfSePZmMqfqsvMCeSsssYt" _
                       "TX7W1DEnbvA+SdWg35zhS4utAYnlAjzJtaqoB4EYmk8x" _
                       "t5DCeNd/vSwTMOhlsXFXYHkxOnO5Va5+etQ1c3j9dOWo" _
                       "O7+Mu6yxzgJnBN6I9lLYK8jbAAAAFQCkjYEHTB8PnKkX" _
                       "UBf2yk+aykSeaQAAAIAe2I7x9TYC9Eas1BqMgZbOBGgX" _
                       "rOjo/a5ZJdFIY22in2t9yAhaqbVbgSpPN9lIDtOab1JG" _
                       "3bzb8Gb9OpvKBiOtMKj4vd8fhUm5SzujJW7sP+FkWixe" _
                       "vi+EnfUFQRIgLTeKKe6QDAPxOUcH84pWKMuxiW9xlcXA" _
                       "JzvuGb2iQQBNLwAAAIAE2tJjK+dJZWoudzvv8pDWWk2H" _
                       "+QxzEGpsCWJQJNVAarY1nCgy5+pbXyX7M9I1FC/fjmaC" _
                       "BwZR//JuYRfo+29LTsCMAk9bOfSrToszXvXgtJ86nWzn" _
                       "1Sz9w3yDgtxpoD8R/mUqa8Xf5J7uGwOT6ypBMa+7u2sG" _
                       "rqD6RiSvCGxGbQ== example";
            }
        }
    }
    call jcs:emit-change( $dot = system, $content, $tag = "transient-change"
);
}
```

Chapter 12
Try It Yourself: MTU Changes

Design a configuration macro with two parameters. The first parameter refers to the desired MTU value and the second is a regular expression for all interfaces that should be assigned the MTU value. Create a commit script that looks for the configuration macro in the `[edit interfaces]` hierarchy and makes the instructed MTU changes in response. The configuration macro should be removed as part of the configuration change.

```
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 {

   /* Allows multiple set-mtu macros to be present */
   for-each( interfaces/apply-macro[ starts-with( name, "set-mtu" ) ] ) {
      var $value = data[name == "value"]/value;
      var $interfaces = data[name == "interfaces"]/value;

      /* Only use if the parameters are present */
      if( jcs:empty( $value ) || jcs:empty( $interfaces ) ) {
         <xnm:warning> {
            call jcs:edit-path();
            <message> "Macro is missing its value and/or interfaces
parameter";
         }
      }
      else {
         /* Scroll through all interfaces that match the regex */
         for-each( ../interface[ jcs:regex( $interfaces, name ) ] ) {

            var $content = {
               <mtu> $value;
            }
            var $message = "Setting MTU to " _ $value;
            call jcs:emit-change( $content, $message );
         }

         /* Remove the instruction macro */
         <change> {
            <interfaces> {
               <apply-macro delete="delete"> {
                  <name> name;
               }
            }
         }
      }
   }
}
```

Chapter 12
Try It Yourself: Custom Firewall Filter

Design a configuration macro that has two parameters, one that indicates the control protocol between PE and CE (BGP, OSPF, etc.), and the other that indicates the policer bandwidth. Create a commit script that transiently creates a firewall filter for each logical interface with that macro configured. The firewall filter should

allow all packets from the control protocol in the first term, and allow all packets in the second term, but rate-limit them to the bandwidth specified in the macro.

```
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 {

   for-each( interfaces/interface/unit/apply-macro[ name == "ingress-
filter" ] ) {
      var $protocol = data[name == "protocol"]/value;
      var $bandwidth = data[name == "bandwidth"]/value;

      /* Only use if the parameters are present */
      if( jcs:empty( $protocol ) || jcs:empty( $bandwidth ) ) {
         <xnm:warning> {
            call jcs:edit-path();
            <message> "Macro is missing its protocol and/or bandwidth
parameter";
         }
      }
      else {
         /* Create filter and policer name */
         var $filter-name = "ingress-filter-" _ ../../name _ "." _ ../name;
         var $policer-name = "ingress-policer-" _ ../../name _ "." _ ../
name;

         /* Assign to interface */
         var $content = {
            <family> {
               <inet> {
                  <filter> {
                     <input> {
                        <filter-name> $filter-name;
                     }
                  }
               }
            }
         }
         call jcs:emit-change( $dot = .., $content, $tag = "transient-
change" );

         /* Create firewall filter and policer */
         <transient-change> {
            <firewall> {
               <family> {
                  <inet> {
                     <filter> {
                        <name> $filter-name;
                        <term> {
                           <name> "allow-control";
                           if( $protocol == "bgp" ) {
                              <from> {
                                 <protocol> "tcp";
                                 <port> "bgp";
                              }
```

```
                                    }
                                    else if( $protocol == "ospf" ) {
                                        <from> {
                                            <protocol> "ospf";
                                        }
                                    }
                                    else { /* RIP */
                                        <from> {
                                            <protocol> "udp";
                                            <port> "rip";
                                        }
                                    }
                                    <then> {
                                        <accept>;
                                    }
                                }
                                <term> {
                                    <name> "police-and-accept";
                                    <then> {
                                        <policer> $policer-name;
                                        <accept>;
                                    }
                                }
                            }
                        }
                    }
                    <policer> {
                        <name> $policer-name;
                        <if-exceeding> {
                            <bandwidth-limit> $bandwidth;
                            <burst-size-limit> "100k";
                        }
                        <then> {
                            <discard>;
                        }
                    }
                }
            }
        }
    }
}
```

What to Do Next & Where to Go ...

http://www.juniper.net/dayone

Get all the Day One books and new This Week titles, too. All from Juniper Networks Books. Check for new automation books as they get published.

http://www.juniper.net/automation

The Junos Automation home page, where plenty of useful resources are available including training class, recommended reading, and a script library - an online repository of scripts that can be used on Junos devices.

http://forums.juniper.net/jnet

The Juniper-sponsored J-Net Communities forum is dedicated to sharing information, best practices, and questions about Juniper products, technologies, and solutions. Register to participate at this free forum.

http://www.juniper.net/techpubs/en_US/junos/information-products/topic-collections/config-guide-automation/frameset.html

All Juniper-developed product documentation is freely accessible at this site, including the Junos API and Scripting Documentation.

http://www.juniper.net/us/en/products-services/technical-services/j-care/

Building on the Junos automation toolset, Juniper Networks Advanced Insight Solutions (AIS) introduces intelligent self-analysis capabilities directly into platforms run by Junos. AIS provides a comprehensive set of tools and technologies designed to enable Juniper Networks Technical Services with the automated delivery of tailored, proactive network intelligence and support services.