



NETWORK ORCHESTRATION WITH Junos XML API and NETCONF

Day One Training and Workshop

IN ONE DAY ...

UNDERSTAND THE JUNOS XML API

- Understand the Junos XML API framework
- Find Junos XML API documentation on the Juniper Networks customer support portal
- Understand Junos configuration XML edit-hierarchy elements (schema)
- Understand Junos operational XML commands and responses (schema)
- Learn to display Junos configurations in XML output from the CLI

INSTALL THE JUNOS NETCONF TOOLKIT

- Download and install the Juniper Networks provided Junos NETCONF Perl toolkit
Orchestrating Junos does not require Perl ... but the toolkit environment does

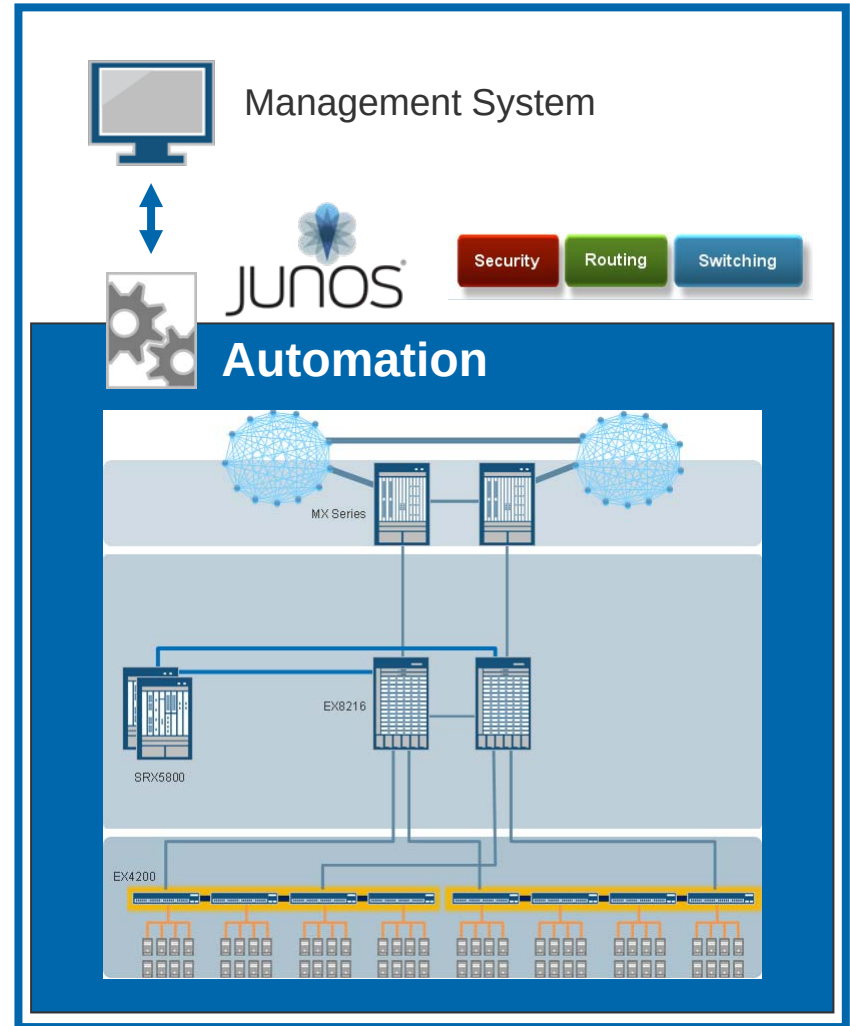
DEVELOP ORCHESTRATION SCRIPTS

- Learn to develop Perl scripts using the Juniper Network toolkit
- Review sample Perl scripts
- Write and test Perl based scripts

UNDERSTANDING THE JUNOS XML API

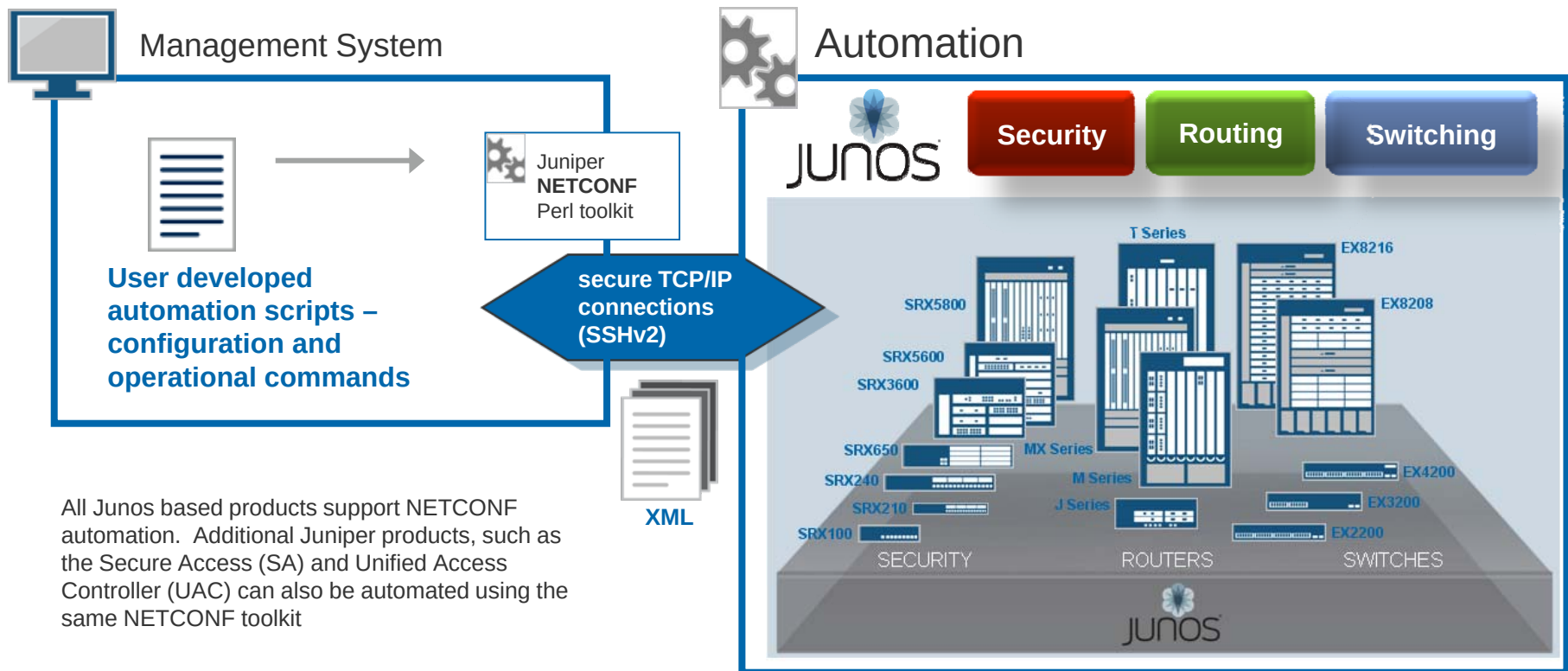
NETWORK ORCHESTRATION

- **ORCHESTRATION** is the *in vogue* term associated with using a single management system to remotely configure multiple devices in order to deliver a seamless service management experience
- **NETWORK ORCHESTRATION** applies this term to the networking devices such as switches, routers, security devices, but not to servers, storage, and applications
- **Junos** supports network orchestration through a published XML API interface



JUNOS “OFF-BOX” AUTOMATION OVERVIEW

Junos “off-box” automation is fast, secure, and reliable. The framework for Junos orchestration is illustrated below. The “off-box” automation scripts (“orchestration”) are located on a management server. These scripts communicate to the Junos devices using the NETCONF protocol. The NETCONF protocol is the best-practices method for network orchestration as it uses XML for structured reliability and SSH for security.



JUNOS XML API AUTOMATION

JUNOS AUTOMATION HERITAGE

- Junos XML API launched in 2001 (Junos 4.2)
- Junos uses XML internally for CLI output
- New features (configurations, commands, etc) are immediately available to the XML API
- Junos leverages the same underlying XML API to enable "on-the-box" automation (*covered in other Day-One training*)

JUNOS XML API

- Well defined XML schema for configuration and operational commands
- Management systems communicates with Junos devices using NETCONF standard (RFC4741)
- NETCONF is a protocol for device management using XML over SSH (secure!)
- Juniper Networks provides a Perl based toolkit to support Junos "off-box" automation development

KEY BENEFITS OF XML API vs. "SCRAPING THE CLI"

- Built for Automation: XML is a structured language built for automation / machine processing
- Future Proof: Changes in the CLI do not break existing automation applications
- Granular and Specific: XML documented schemas for configuration and operational commands
- Reliable Error Handling: Structured XML error response values vs. processing spurious CLI results
- Enables Automation Engineering: XML response processing very easy to handle given extensive industry tools, such as XPath navigation

JUNOS XML SCHEMA OVERVIEW

CONFIGURATION

- Junos configuration is declared between the <configuration> element block. There can be only one <configuration> element block in an XML document
- Within the <configuration> block there can be any number of Junos “top-level” element blocks; e.g. multiple <interfaces>, <system>, <vlans>, etc.
- The Junos XML configuration schema is documented:
“JUNOS XML API Configuration Reference”

OPERATIONAL COMMANDS

- Junos operational commands are declared in unique element blocks with well defined parameters. For example, the CLI “show route” command first element is:
<get-route-information>
- The Junos XML operational command schema is documented:
“JUNOS XML API Operational Reference”

JUNIPER WEBSITE:

- <http://www.juniper.net/techpubs/software/junos/index.html>
- Select “JUNOS API and Scripting Documentation”

JUNOS XML API

CONFIGURATION EXAMPLE

The following illustrates an XML document that would add a VLAN named “Accounting” to a Junos device. The corresponding CLI rendered output is also shown

XML

```
<configuration>
  <vlans>
    <vlan>
      <name>Accounting</name>
      <description>Accounting Department</description>
      <vlan-id>500</vlan-id>
      <interface>
        <name>ge-0/0/12.0</name>
      </interface>
      <interface>
        <name>ge-0/0/13.0</name>
      </interface>
      <interface>
        <name>ae0.0</name>
      </interface>
    </vlan>
  </vlans>
</configuration>
```

CLI

```
[edit]
admin@EX-dc1# show vlans Accounting
description "Accounting Department";
vlan-id 500;
interface {
  ge-0/0/12.0;
  ge-0/0/13.0;
  ae0.0;
}
```

The Junos CLI can display the XML configuration using the “| display xml” option

```
[edit]
admin@EX-dc1# show vlans Accounting | display xml
```

JUNOS XML API

OPERATIONAL COMMAND EXAMPLE

The following XML document illustrates the “show chassis hardware detail” operational command.

XML COMMAND

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

XML RESPONSE

```
<chassis-inventory>
  <chassis>
    <name>Chassis</name>
    <serial-number>AD2909AA0096</serial-number>
    <description>SRX210-hm</description>
    <chassis-module>
      <name>Routing Engine</name>
      <version>REV 28</version>
      <part-number>750-021779</part-number>
      <serial-number>AAAH4755</serial-number>
      <description>RE-SRX210-HIGHMEM</description>
      <chassis-re-usb-module>
        <name>usb0 (addr 1)</name>
        <product>DWC OTG root hub</product>
        <product-number>0</product-number>
      </chassis-re-usb-module>
    </chassis-module>
  </chassis>
</chassis-inventory>
```

The Junos CLI can display the XML operational response using the “| display xml” option. As of Junos 10.1, the “| display xml rpc” option will show the XML command.

```
admin@SRX-dc1> show chassis hardware detail | display xml rpc
```

JUNOS XML API PROTOCOL METHODS

DEVICE MANAGEMENT PROTOCOLS

- There are two protocol methods for communicating with Junos devices: NETCONF and Junos XML protocol (fka JUNOScript)
- Both methods use the same Junos XML schema definitions for configuration and operational commands
- NETCONF protocol: RFC-4741 is a standards based approach for remotely managing devices via XML over SSH
- Juno XML protocol: Juniper Networks original specification which was used as a basis for NETCONF

WHICH PROTOCOL METHOD TO USE?

- NETCONF should be used as it is standards based and supports all Junos functionality
- NETCONF uses SSH for secure device access. If other access methods are required (TELNET, SSL, clear-text) then Junos XML protocol can be used
- The following charts provide use comparisons between NETCONF and Junos XML protocol that can help determine which protocol is best for the end customer automation environment

COMPARING NETCONF WITH JUNOS XML PROTOCOLS

CONTINUED

The following chart compares the commonly used configuration capabilities between the two protocol methods:

CONFIGURATION	NETCONF protocol	Junos XML protocol
Add configuration sections	✓	✓
Delete configuration sections	✓	✓
Rollback configuration (and compare)	✓	✓
Activate / Deactivate configuration sections	✓ *	✓
Re-order configuration sections using “insert”	✓ *	✓
Rename configuration sections	✓ *	✓
Retrieve “Rescue” configuration	✓	✓
“commit check”	✓	✓

*** NOTE:** The current NETCONF guide does not describe the XML attributes for “insert”, “rename”, “activate”, and “deactivate”. The Junos NETCONF implementation supports the same attributes as described in the JUNOS XML Protocol guide

COMPARING NETCONF WITH JUNOS XML PROTOCOLS

The following chart compares the access method between the two protocol methods:

ACCESS METHODS	NETCONF protocol	Junos XML protocol
SSH	✓	✓
Outbound SSH	✓	✓
Secure Socket Layer (SSL)	✗	✓
TELNET	✗	✓
Clear-Text	✗	✓

The **Outbound SSH** feature allows the initiation of an SSH session between routers running Junos Software and Network and System Management servers where client-initiated TCP/IP connections are blocked (for example, when the router is behind a firewall).

Junos XML protocol provides alternatives if SSH cannot be used. A client application that uses the Junos XML protocol-specific **clear-text** access method sends unencrypted text directly over a TCP connection without using any additional transport protocol (such as SSH, SSL, or Telnet).

LAB EXERCISES:

1. Login to a Junos device. From the CLI run operational commands and display the output in XML form
2. Login to a Junos device. From the CLI run operational commands and display the command in XML form
3. Login to a Junos device. From the CLI enter configuration mode, and display sections of the configuration in XML form

INSTALL JUNOS NETCONF PERL TOOLKIT

JUNOS XML AND NETCONF TOOLKIT

DOWNLOAD JUNOS XML API DOCUMENTATION [OPTIONAL]

<https://www.juniper.net/support/xml/junos-xml/index.html>

- Junos configuration schema
- Junos operational command schema
- Junos Document Type Definitions (DTDs)

DOWNLOAD NETCONF API PERL TOOLKIT

<https://www.juniper.net/support/xml/netconf/index.html>

- Device Drivers + Example Scripts
- Supporting Libraries/Perl Modules (“prerequisites”)

INSTALLING NETCONF PERL TOOLKIT

INSTALLATION OVERVIEW

- There are two “tarball” files that makeup the toolkit: the main driver tarball, and the “prerequisite” tarball
- The prereq tarball contains the system libraries and additional Perl modules that are used by the Juniper toolkit/driver. The current “prereq” tarball was built a number of years ago (2007), and has not been recently updated
- Recommendation would be to check your current system to determine which libraries and Perl modules are needed, and download the most recent ones; i.e. do not use the files in the “prereq” tarball
- The recommended installation steps are described on the following slide
- Confirm your management system has Internet access
- Confirm your management system has the “cpan” tool installed
- Confirm you are familiar with obtaining OS package files (e.g. “rpm”) for libraries that you need
- Confirm you have the following “build” tools installed on your systems: gcc/g++, patch, m4, and binutils

INSTALLING NETCONF PERL TOOLKIT CONTINUED

UNPACK THE NETCONF TARBALL (e.g. netconf-perl-n.n.tar.gz)

- Once you've unpacked the netconf tarball, cd into the directory and unpack the "prereq" tarball
- Run the command:
`> perl required-mod.pl`
- You will most likely see a list of needed items. For each item, use the cpan tool to download the latest version
- Recent installations have also shown the following Perl modules needed to be installed. You can use the cpan tool to download these modules and the supporting libraries:

XML::LibXML

Math::BigRat

Math::BigInt::GMP

Math::BigInt::Pari

Net::SSH::Perl

- Once all of the "required-mod" files have been installed, run the following commands from the netconf directory:
`> perl Makefile.PL`
`> make test`
`> make install`

RUNNING EXAMPLE SCRIPT

TEST USING SAMPLE SCRIPT

- Confirm that your management system can access the Junos device (“ping”)
- Confirm that you have a login/password on the Junos device
- Confirm that the device is enabled for NETCONF SSH

```
[edit system services]
admin@SRX-dc1# show
ssh;
netconf {
    ssh;
}
```

- For security devices (e.g. “SRX”) you will need to ensure that SSH is an enabled inbound host service

```
[edit security zones functional-zone management]
admin@SRX-dc1# show
interfaces {
    fe-0/0/5.0;
}
host-inbound-traffic {
    system-services {
        ssh;
        netconf;
    }
}
```

RUNNING EXAMPLE SCRIPT CONTINUED

- From management system
 - > cd examples
 - > cd get_chassis_inventory
 - > perl get_chassis_inventory.pl <hostname-ip>
- You will be prompted for a login and password
- The script should take a few seconds to run. The successful output will look something like the following:

```
$ perl get_chassis_inventory.pl 192.168.2.14
login: admin
password:
192.168.2.14.xml:2: namespace error : Namespace prefix junos for style on
chassis is not defined
<chassis junos:style="inventory">
      ^
CHASSIS,MODULE,SUBMODULE,VERSION,PARTNUMBER,SERIALNUMBER,DESCRIPTION
Chassis,,,,,AD2909AA0096,SRX210-hm
Chassis,Routing Engine,,REV 28,750-021779,AAAH4755,RE-SRX210-HIGHMEM
Chassis,FPC 0,,,,,FPC
Chassis,FPC 0,PIC 0,,,,,2x GE, 6x FE, 1x 3G
Chassis,FPC 1,,REV 04,750-023367,AAAG7981,FPC
Chassis,FPC 1,PIC 0,,,,,1x T1E1 mPIM
Chassis,Power Supply 0,,,,,
```

NOTE: You will notice a “namespace” error in the output. You can ignore this as it is a result of the management system and not a problem with the toolkit.

LAB EXERCISES:

1. Download Junos NETCONF toolkit
2. Install toolkit onto management system
3. Confirm NETCONF SSH access on Junos device
4. Test environment using the `get_chassis_inventory` sample



DEVELOP CONFIGURATION SCRIPTS

THREE EASY STEPS ...

STEP 1 – CAPTURE XML TEMPLATE

Login to the Junos device and create a working configuration. Once you have a working “template”, display the XML configuration and save it for reference.

STEP 2 – AUTOMATE XML DOCUMENT

Create a Perl script using the XML template as a reference. The Perl script should generate the XML document with the required elements and tags supplemented with the appropriate automation variables.

STEP 3 – TRANSFER XML CONFIGURATION TO DEVICE

Use NETCONF to transfer the XML document to the device. The management system will need to perform the necessary lock/unlock calls around committing the configuration.

DEVELOP JUNOS XML DOCUMENTS

WITH **EzEditXML** PERL MODULE

EzEditXML

WHAT IS EZEDITXML?

- EzEditXML is a Perl module that simplifies the development of XML documents (step-2). In programming parlance it is “syntactic sugar”
- EzEditXML can be used to produce XML documents for both configuration and operational command operations
- EzEditXML provides Junos CLI specific features to manipulate the configuration, corresponding to the CLI commands: delete, activate, deactivate, insert, and rename
- EzEditXML uses XML::LibXML as a base library. XML::LibXML is the “state-of-the-art” in terms of XML processing for Perl environments and offers very powerful programmatic capabilities both in terms of creating and parsing XML documents

WHY USE EZEDITXML?

- EzEditXML makes developing Junos XML fast and easy, reducing the overall time to develop Junos automation
- Alternatively the scripts could be coded in “raw” XML. This approach, however, has many drawbacks when the automation scripts tend to get more complex or require programmatic logic processing

WHY NOT SIMPLY USE EMBEDDED CLI COMMANDS?

- NETCONF support the ability to configure the Junos device using CLI commands. That is CLI text, e.g. “set system host-name SRX1-dc1”, can be embedded in the NETCONF protocols
- This technique may be suitable for small and static configuration templates. Complex templates that require many variables and/or the use of programming logic do not lend themselves well to the CLI model (but are workable)

“HELLO, WORLD!”

The following example will produce the XML configuration document that would set the Junos device host - name to “SRX210_lab”.

```
#!/perl  
  
use JUNOS::EzEditXML;  
  
$config = new JUNOS::EzEditXML;  
  
$config->system->host_name( 'SRX_lab1' );  
  
print $config->toString;
```

```
<configuration>  
  <system>  
    <host-name>SRX210_lab1</host-name>  
  </system>  
</configuration>
```

It is important to note that EzEditXML is not required for using the Junos XML API or the NETCONF toolkit provided by Juniper Networks. EzEditXML is one tool that can quickly, and easily facilitate the development of XML documents (configurations, operational commands) to be used with the NETCONF toolkit. Depending on your specific automation needs, other simpler methods could be used in place of, or in conjunction with, the EzEditXML package.

EzEditXML is only used to create XML documents. It does not perform the device communication with the Junos targets. That function is performed by the NETCONF toolkit library, presented in the next section.

EXAMPLES:



Vlans.pl



System.pl



Security.pl



interfaces.pl



Inetaddr.pl



Groups.pl

EzEditXML METHODS

METHOD	DESCRIPTION
new	Constructor used to create the “top” EzElement that corresponds to the the first XML element in the document; e.g. <configuration> for Junos configurations
<i>AUTOLOAD</i>	EzEditXML utilizes Perl’s package AUTOLOAD method to dynamically create XML elements and associated child-elements. The vast majority of EzEditXML scripts can solely use the new, AUTOLOAD, and toString methods. The AUTOLOAD will return the newly created or existing EzElement
addPath	Used to create a collection of XML elements in an associated “edit path” order. Child elements for each directory in the path can also be assigned.
newElement	Creates a new and unattached EzElement. Children are added if provided.
addElement	Adds a new/unattached EzElement to the XML document, or creates a new EzElement depending on the parameters. Children are added if provided.
addXML	This method takes a XML well-balanced-chunk and incorporates/attaches it into an EzEditXML document. This is very handy for automating static chunks of XML.
addSibling	Used to create a new EzElement at the same XML hierarchy with the same starting element name
setText	Used to change the element text-node value.
toString	Used to return the XML document in string form
getLibNode	Used to return the XML::LibXML::Node associated with the EzElement. This is very useful when you want to utilize the XML::LibXML libraries.

EzEditXML JUNOS SPECIFIC METHODS

METHOD	DESCRIPTION
junosDelete	Used to mark the given element block (section) for delete. This is the equivalent of the CLI command “# delete ...”
junosActivate	Used to mark the given element block to activate. This is the equivalent of the CLI command “# activate ...”
junosDeactivate	Used to mark the given element block to deactivate. This is the equivalent of the CLI command “# deactivate ...”
junosInsert	Used to mark the given element block to relocate (insert). This is the equivalent of the CLI command “# insert ...”
junosRename	Used to mark the given element block to be renamed. This is the equivalent of the CLI command “# rename ...”

It is important to note that these methods only *mark* the XML document (i.e. set XML attributes) to perform a given action. The Junos configuration on the target device will be modified once the configuration is transmitted to the device and the configuration is committed. The NETCONF toolkit driver section describes the device communication process and methods.

CONSTRUCTOR

new

```
$ezElem = new EzEditXML( [$stopTag] )
```

- [\$stopTag] is optional. Default value will create a top-tag of <configuration>
- Returns object that is used to construct XML document
- Use toString method to emit XML document in string form

CONFIGURATION EXAMPLE

```
use JUNOS::EzEditXML;  
  
$config = new JUNOS::EzEditXML;  
  
$config->system->host_name( 'SRX_lab1' );  
  
print $config->toString;
```

```
<configuration>  
  <system>  
    <host-name>SRX210_lab1</host-name>  
  </system>  
</configuration>
```

OPERATIONAL COMMAND EXAMPLE

```
use JUNOS::EzEditXML;  
  
$op_cmd = new JUNOS::EzEditXML('get-route-information');  
  
$op_cmd->destination('192.168/16');  
  
print $op_cmd->toString;
```

```
<get-route-information>  
  <destination>192.168/16</destination>  
</get-route-information>
```

AUTOLOAD USING OBJECT-ORIENTED SYNTAX (->) OVERVIEW

When a method name is “unknown” to Perl will automatically create it using the Perl AUTOLOAD module method. EzEditXML uses the AUTOLOAD to dynamically create XML elements and parent/child structures.

```
1. use JUNOS::EzEditXML;
2. $config = new JUNOS::EzEditXML;
3. $sys = $config->system;
4. $sys->host_name('SRX-dc1');
5. $svcs = $sys->services;
6. $svcs->ssh;
7. $svcs->web_management->http->interface( 'ge-0/0/12.0' );
```

Line 3 creates a new section of the XML configuration for the system hierarchy. We assign the return value of the method `system` to the `$sys` to act as a shortcut when we later add XML elements.

Line 4 adds a new XML element called `host - name` to the XML structure. Since `host_name` did not exist, then the AUTOLOAD feature kicked in and created a new method called `host_name`. The method `host_name` creates an XML child element called `host - name` (converting “_” to “-”). Since a SCALAR value was passed to method `host_name`, the method uses that parameter to assign the XML element text node, resulting in:

```
<system>
  <host - name>SRX-dc1</host - name>
</system>
```

AUTOLOAD USING OBJECT-ORIENTED SYNTAX (->) CONTINUED

```
1. use JUNOS::EzEditXML;
2. $config = new JUNOS::EzEditXML;
3. $sys = $config->system;
4. $sys->host_name('SRX-dc1');
5. $svcs = $sys->services;
6. $svcs->ssh;
7. $svcs->web_management->http->interface( 'ge-0/0/12.0' );
```

Line 5 creates a new child services to system (via the \$sys object). Again since services does not exist, the AUTOLOAD routine kicks in, creates a new method, which in turn creates a new XML element (services) and attaches it to system. We can assign a new handle for services called \$svcs to act as a shortcut when we want to add additional items below services. As we continue to dynamically build parent/child relationships in this manner, the XML hierarchical document is constructed.

Line 6 invokes method ssh, it doesn't exist, creates the XML, and since there is nothing more done with ssh, that is the final child node in the structure.

Line 7 invokes method web_management, which creates the XML element web-management. web-management invokes http, which creates the XML element http. http invokes interface, which creates the XML element interface. Since a scalar argument was passed to method interface, the given text is assigned to the interface element.

The complete resulting XML is shown again on the next slide ...

AUTOLOAD USING OBJECT-ORIENTED SYNTAX (->) CONTINUED

```
use JUNOS::EzEditXML;  
  
$config = new JUNOS::EzEditXML;  
$sys = $config->system;  
  
$sys->host_name('SRX-dc1');  
  
$svcs = $sys->services;  
$svcs->ssh;  
$svcs->web_management->http->interface('ge-0/0/12.0');
```

```
<system>  
  <host-name>SRX-dc1</host-name>  
  <services>  
    <ssh/>  
    <web-management>  
      <http>  
        <interface>ge-0/0/12.0</interface>  
      </http>  
    </web-management>  
  </services>  
</system>
```

It is important to note that EzEditXML is not checking the dynamic method names to a Junos dictionary. This means that any XML structure can be created. So take care that the method names map to the specific Junos configuration hierarchy values that you are trying to automate!

AUTOLOAD ELEMENT WITH ASSIGNING TEXT-NODE VALUE

```
$newEzElem = $ezElem->ELEMENT( $textValue )
```

- \$textValue is a string value that is used to set the XML element text-node value
- The newly created ELEMENT in EzElement form is returned

The following example illustrates this technique. The ELEMENT in this example `host_name` and the scalar value is `SRX-dc1`

```
$config->system->host_name( 'SRX-dc1' );
```

Resulting XML:

```
<system>  
  <host-name>SRX-dc1</host-name>  
</system>
```

It is important to note that subsequent calls to do the same will *append* text to the ELEMENT and not replace the value. To replace the text-value, use the `setText` method.

AUTOLOAD ELEMENT WITH LIST OF “LIKE” CHILDREN

`$ezElem->ELEMENT( $arefText, [$options] )`

- `$arefText` is an array-reference storing a scalar list. This technique is used to create a repetitive list of “ELEMENT” for each of the scalars
- `$options` are optional arguments used to perform modifications. These include the following and can be combined in a comma separated list:
 - `nameItems=>1` is used to add a name child element for each scalar item
 - `junosDelete=>1` is used to mark the scalars for deletion
 - `junosActivate=>1` is used to mark the scalars for activation
 - `junosDeactivate=>1` is used to mark the scalars for deactivation

The following example illustrates this technique. The ELEMENT in this example is application and the scalar values are `junos-ftp`, `junos-ntp`, and `junos-ssh`.

```
$sec = $config->addPath( 'security/policies/policy',  
    policy => [{'from-zone-name' => 'trust'},  
               {'to-zone-name' => 'untrust'}] );  
  
$pol = $sec->policy->policy;  
$pol->name( 'my-sec-policy' );  
$pol->match->application( ['junos-ftp', 'junos-ntp', 'junos-ssh'] );  
$pol->match->source_address( 'any' );  
$pol->match->destination_address( 'any' );  
$pol->then->permit;
```

AUTOLOAD

ELEMENT WITH LIST OF “LIKE” CHILDREN CONTINUED

```
$pol->match->application( ['junos-ftp', 'junos-ntp', 'junos-ssh'] );
```

Resulting XML:

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

AUTOLOAD

ELEMENT WITH LIST OF “LIKE” CHILDREN CONTINUED

The Junos XML schema commonly employs name child-elements to uniquely identify the element configuration. A good example of this usage is found in the interfaces defined within a VLAN. Each interface element contains a name child that identifies the specific interface name, as shown:

```
<vlans>
  <vlan>
    <name>Blue</name>
    <vlan-id>20</vlan-id>
    <l3-address>vlan.20</l3-address>
    <interface>
      <name>ge-0/0/0.0</name>
    </interface>
    <interface>
      <name>ge-0/0/1.0</name>
    </interface>
    <interface>
      <name>ge-0/0/2.0</name>
    </interface>
  </vlan>
</vlans>
```

To automate the addition of the name child-element, use the `nameItems=>1` optional argument, as showing in the example on the next slide ...

AUTOLOAD

ELEMENT WITH LIST OF “LIKE” CHILDREN CONTINUED

The following example illustrates this technique. The ELEMENT in this example is `interface` and the `nameItems=>1` option is used to generate the name child-element for each scalar value in the array.

```
$vlans = $config->addPath( 'vlans/vlan',  
    vlan => [{ 'name' => 'Blue'},  
             { 'vlan-id' => '20'},  
             { 'l3-address' => 'vlan.20' }]);  
  
$vlan = $vlans->vlan;  
$vlan->interface( ['ge-0/0/0.0', 'ge-0/0/1.0', 'ge-0/0/2.0'], nameItems=>1 );
```

```
<vlans>  
  <vlan>  
    <name>Blue</name>  
    <vlan-id>20</vlan-id>  
    <l3-address>vlan.20</l3-address>  
    <interface>  
      <name>ge-0/0/0.0</name>  
    </interface>  
    <interface>  
      <name>ge-0/0/1.0</name>  
    </interface>  
    <interface>  
      <name>ge-0/0/2.0</name>  
    </interface>  
  </vlan>  
</vlans>
```

AUTOLOAD ELEMENT WITH LIST OF UNIQUE CHILDREN

```
$newElem = $ezElem->ELEMENT( $arefHash )
```

- \$arefHash is an array-reference storing a list of hash-references. This technique is used to add a collection of child-elements to the parent ELEMENT

```
$addr = $inet->address;    # shortcut to the 'address' element
```

```
$addr->vrrp_group( [{ 'name' => '10'},  
                  { 'priority' => '200'},  
                  { 'virtual-address' => '192.168.13.20'} ] );
```

```
<interfaces>  
  <interface>  
    <name>ge-0/0/0</name>  
    <unit>  
      <name>1920</name>  
      <family>  
        <address>  
          <name>192.168.12.23/24</name>  
          <vrrp-group>  
            <name>10</name>  
            <priority>200</priority>  
            <virtual-address>192.168.20.20</virtual-address>  
          </vrrp-group>  
        </address>  
      </family>  
    </unit>  
  </interface>  
</interfaces>
```

The above example illustrates this technique. The ELEMENT is vrrp_group and the child-elements are name, priority, and virtual-address. The child-elements will be added in the order they are provided in the array.

addChildren

```
$newEzElem = $ezElem->addChildren( $children )
```

- This method is the “workhorse” of the EzEditXML module. You have already seen the various options for defining children elements in the AUTOLOAD section. To recap, there are three forms for `$children`:
 - `$textValue` – Assigns the `$ezElem` text-node value
 - `$arefText` – Adds a list of “like” children items with the same element-name as `$ezElem`. The list of children are added to the parent of `$ezElem`; that is the `$ezElem` acts at the common name for each of the list item. This form also includes a comma separated option list as described on slide-32.
 - `$arefHash` – Adds a list of unique children under the `$ezElem` element-block
- For EXAMPLES, please refer back to the AUTOLOAD slides as well as the next slides on the `addPath` method

addPath

```
$newEzElem = $ezElem->addPath( $path, [$children])
```

- `$path` is a string representing the Junos edit-hierarchy, indicating parent/child relationship.
Examples:
 - `'system/services'`
 - `'interfaces/interface/family/unit/address'`
 - `'vlans/vlan'`
- `$path` does not need to start at the “top-level” of the Junos configuration hierarchy. It can be any relative “path” from `$ezElem`
- The resulting XML formulated by the `$path` and `$children` is attached at `$ezElem`
- `$newEzElem` is the element associated with the first item in the `$path`. For example, if the `$path = 'interface/unit/inet/address'`, then `$newEzElem` is bound to the interface element
- `$children` is an optional set of hashes (key/value) pairs that allows setting of child-elements within each of the path levels. The format of `$children` is the same as documented in the `addChildren` method

addPath

WITH CHILDREN EXAMPLES

The following example produces the XML that sets the system host - name element to the value SRX210_lab1. Note if the element name contains a dash ('-') then the hash-key must be surrounded by quotes, as shown:

```
$config->addPath( 'system/host-name', 'host-name' => 'SRX210_lab1' );
```

The following example produces the XML that creates a VLAN with child elements name FOO-vlan and a VLAN - ID of 200. Note how the two child elements to the `vlan` are contained within an array reference (`[ ... ]`). Each child is stored in a hash (`{key => value}`). The order of the hashes in the array is significant; the children are added in the order they are listed in the array, as shown:

```
$config->addPath( 'vlans/vlan',  
  vlan => [{ 'name' => 'FOO-vlan'}, { 'vlan-id' => 200 } ] );
```

The following example produces the XML that configures an interface inet address. Note how writing XML elements and children simultaneously is also possible, as shown:

```
$config->addPath( 'interfaces/interface/unit/family/inet/address',  
  interface => [{ 'name' => 'ge-0/0/27' }],  
  unit => [{ 'name' => 0 }],  
  address => [{ 'name' => '192.168.13.57/24' } ] );
```

newElement

```
$newEzElem = $ezElem->newElement( $element, [$children] )
```

- This method is used to create an unattached EzElement object. This can be a handy technique if you need to create “chunks” of XML documents and attach them at a later time
- See EXAMPLES on next slide ...

addElement

```
$newEzElem = $ezElem->addElement( $elemName, [$children])
```

- This form of the addElement method is used to create a new EzElement and attach it under \$ezElem. In this form, the addElement method calls newElement first to create the EzElement and then attach it to \$ezElem. EXAMPLE:



Security.pl

```
$newEzElem = $ezElem->addElement( $ezElem, [$children])
```

- This form on the addElement method is used to attach an existing EzElement that was created using the newElement method. EXAMPLE:



System.pl

addXML

`$ezElem->addXML( $xmlstring )`

- This method is used to add a “well-balanced” XML string to the EzEditXML document, attached at `$ezElem`. A “well-balanced” XML string means that all element-tags have the correct open tags and close-tags
- This method is typically used to include static blocks of XML configuration that does not require any variables / automation
- EXAMPLE:



Groups.pl

addSibling

```
$newEzElem = $ezElem->addSibling
```

- This method is used to create a new EzElement at the same XML document level with the same starting element name \$ezElem
- This method is typically used when creating multiple blocks of the same type, such as security policies, firewall policies, routing policies, etc.

setText

```
$ezElem->setText( $newText )
```

- This method is used to change the text-value for a given element to \$newText

toString

`$string = $ezElem->toString`

- Used to return the XML document in string form. The `$ezElem` is typically the variable returned from the new method. However the `toString` method will work on any of the `EzElement` objects

junosDelete

```
junosDelete $ezElem  
$ezElem->junosDelete
```

This method is used to delete the designated configuration. It is important to know which element to mark for deletion. In the case of elements with a name child-element, the parent (not name) needs to be marked for deletion.

The `junosDelete=>1` optional parameter performs this same function (*see prior slides*)

The following is equivalent to the CLI command “`delete vlans Blue`”

```
$vlan = $config->vlans->vlan;  
$vlan->name( 'Blue' );  
junosDelete $vlan;
```

Resulting XML:

```
<vlans>  
  <vlan delete="delete">  
    <name>Blue</name>  
  </vlan>  
</vlans>
```

junosActivate

```
junosActivate $ezElem  
$ezElem->junosActivate
```

This method is used to activate the designated configuration. It is important to know which element to mark for activation. In the case of elements with a name child-element, the parent (not name) needs to be marked for activation.

The `junosActivate=>1` optional parameter performs this same function (*see prior slides*)

The following is equivalent to the CLI command “activate vlans Blue”

```
$vlan = $config->vlans->vlan;  
  
$vlan->name( 'Blue' );  
junosActivate $vlan;
```

Resulting XML:

```
<vlans>  
  <vlan active="active">  
    <name>Blue</name>  
  </vlan>  
</vlans>
```

junosDeactivate

```
junosDeactivate $ezElem  
$ezElem->junosDeactivate
```

This method is used to deactivate the designated configuration. It is important to know which element to mark for deactivation. In the case of elements with a name child-element, the parent (not name) needs to be marked for deactivation.

The `junosDeactivate=>1` optional parameter performs this same function (*see prior slides*)

The following is equivalent to the CLI command “`deactivate vlans Blue`”

```
$vlan = $config->vlans->vlan;  
  
$vlan->name( 'Blue' );  
junosDeactivate $vlan;
```

Resulting XML:

```
<vlans>  
  <vlan inactive="inactive">  
    <name>Blue</name>  
  </vlan>  
</vlans>
```

junosRename

```
junosRename $ezEdit ( $newName )  
$ezEdit->junosRename( $newName )
```

This method is used to rename the designated configuration element to \$newName. It is important to know which element to rename. In the case of elements with a name child-element, the parent (not name) needs to be renamed.

The following is equivalent to the CLI command “rename vlans Blue to Green”

```
$vlan = $config->vlans->vlan;  
$vlan->name( 'Blue' );  
junosRename $vlan ( 'Green' );
```

Resulting XML:

```
<vlans>  
  <vlan rename="rename" name="Green">  
    <name>Blue</name>  
  </vlan>  
</vlans>
```

junosInsert

```
junosInsert $ezElem( $location, $reference )  
$ezElem->junosInsert( $location, $reference )
```

- `$location` = 'before' or 'after' identifies the position relative to `$reference`
- `$reference` identifies the reference element

This method is used to move the location of the ELEMENT in the configuration relative to the `$reference` reference element.

The following is equivalent to the CLI command:

```
insert policy my-sec-policy before policy my-last-policy
```

Note: This example based on larger script shown on earlier slide, abbreviated here

```
$pol = $sec->policy->policy;  
$pol->name('my-sec-policy');  
$pol->junosInsert( 'before', 'my-last-policy');
```

Resulting XML:

```
<policy insert="before" name="my-last-policy">  
  <name>my-sec-policy</name>  
</policy>
```

LAB EXERCISES:

1. Create a Junos configuration script that performs basic device configuration
2. Create a Junos configuration script that creates a VLAN with a few interfaces
3. Create a Junos configuration script that removes one of the interfaces from the VLAN
4. Create a Junos configuration script that deactivates the VLAN
5. Create a Junos configuration script that activates the VLAN

JUNOS DEVICE COMMUNICATION WITH NETCONF DRIVER MODULE

“HELLO, WORLD!”

The following Perl code highlights all of the necessary routines to create an XML configuration and load it onto the Junos device using the NETCONF driver module. Error handling is not shown to reduce clutter, but is shown in examples on following slides.

```
use JUNOS::EzEditXML;
use Net::Netconf::Manager;

$config = new JUNOS::EzEditXML;

$config->system->host_name('SRX_lab1');

$xml_config = $config->toString;

$jnx = new Net::Netconf::Manager(
    access => 'ssh',
    login => $login,
    password => $password,
    hostname => $hostname );

$jnx->lock_config( target => 'candidate' );

$jnx->edit_config( target => 'candidate', config => $xml_config );

$jnx->commit();

$jnx->unlock_config();

$jnx->disconnect();
```

NETCONF DRIVER METHODS

METHOD	DESCRIPTION
new	Creates a new NETCONF device object and connects to the target device
lock_config	Locks the configuration database (exclusive)
unlock_config	Unlocks the configuration database
edit_config	Edits the candidate configuration
get_config	Retrieves the configuration with option to filter
commit	Commits the candidate configuration
disconnect	Disconnects from the target device
kill_session	Used to remotely kill the NETCONF session with the provide process id
copy_config	Replaces the configuration with the provided XML document or FILE (URL reference)
discard_changes	Performs the equivalent of “rollback 0”
validate	Performs the equivalent of “commit check”

Each of these methods has a corresponding XML tag description documented in the Juniper NETCONF guide. The following slides illustrate the API to most of these routines with example code and commentary.

NEW

```
$hdl = new Net::Netconf::Manager( $args )
```

- This method is used to create a NETCONF device object and connect to the target Junos device via SSH.
- `$args` is a hash containing the following parameters:
 - `hostname` – Hostname or IP-address of Junos device
 - `login` – User account used to access the device
 - `password` – Password associated with <login> user
 - `access` – Must be set to 'ssh'
- Returns `$hdl` object that is used to communicate with Junos target. If this method fails, `$hdl` will be set to `undef`.

NEW EXAMPLE

```
# -----  
# create a NETCONF device object and connect to the Junos device.  If  
# this operation fails, then the return variable is 'undef'  
# -----  
  
$jnx = new Net::Netconf::Manager(  
    access => 'ssh',  
    login => $login,  
    password => $password,  
    hostname => $hostname );  
  
if(! $jnx ) {  
    print STDERR "Unable to connect to Junos device $hostname.\n";  
    exit 1;  
}
```

NOTE: The access parameter must be specified, and it must be 'ssh'.

NETCONF UTILITY METHODS

`get_session_id`

- This method returns the NETCONF session (process) ID for the current connection

`has_error`

- Returns TRUE if the last NETCONF operation failed

`get_first_error`

- Returns the first RPC error response in a Perl hash

`get_all_errors`

- Returns all RPC error response in a Perl hash

lock_config

```
$rply = lock_config( target => 'candidate' )
```

- This method is lock the candidate configuration. It is equivalent to “configure exclusive”
- Returns the NETCONF reply result starting with <rpc-reply ...> . This is generally ignored
- The method has_error () should be checked to determine if this method failed

```
$rc = $jnx->lock_config( target => 'candidate' );  
  
if( $jnx->has_error ) {  
    $errmsg = NETCONF_util_errstr( $jnx );  
    print "ERROR locking config: $errmsg\n";  
    NETCONF_util_graceful_shutdown( $jnx );  
    exit 1;  
}
```

NOTE: The NETCONF_util routines are not part of the NETCONF Perl toolkit, but they are shown here to indicate a typical action. Sample “NETCONF utility” code can be found in the examples directory of the Perl toolkit.

unlock_config

`$rply = unlock_config`

- This method is used to unlock the configuration database
- Returns the NETCONF reply result starting with `<rpc-reply ...>` . This is generally ignored
- The method `has_error()` should be checked to determine if this method failed

```
$rc = $jnx->unlock_config();  
  
if( $jnx->has_error ) {  
    $errmsg = NETCONF_util_errstr( $jnx );  
    print "ERROR unlocking config: $errmsg\n";  
    NETCONF_util_graceful_shutdown( $jnx );  
    exit 1;  
}
```

edit_config

```
$rply = edit_config( $args )
```

- This method is load configuration to the target device. The arguments used for loaded XML based configurations are documented here
- \$args is a hash containing the following parameters:
 - target – always set to ‘candidate’
 - config – The XML configuration document in string form. This document must begin with the <configuration> element
- Additional options to this routine are available that include loading a configuration based on CLI commands (not XML), loading a configuration file (XML or CLI) based on a URL, and other error handling options. These capabilities are fully described in the NETCONF guide in the “Summary of NETCONF Tag Elements” chapter
- Note that not all of the documented capabilities may not be currently supported in the Perl toolkit. Please see section “Customizing the NETCONF Driver” for details on making enhancements and customizations

edit_config EXAMPLE

```
# -----  
# transfer the XML configuration to the Junos device  
# -----  
  
$rc = $jnx->edit_config( target => 'candidate', config => $XML_config );  
  
if( $jnx->has_error ) {  
    $errmsg = NETCONF_util_errstr( $jnx );  
    print "ERROR loading config: $errmsg\n";  
    NETCONF_util_graceful_shutdown( $jnx );  
    exit 1;  
}
```

NOTE: The target parameter must be specified, and it must be 'candidate'.

commit

`$rply = commit`

- This method is used to commit the candidate configuration
- Returns the NETCONF reply result starting with `<rpc-reply ...>` . This is generally ignored
- The method `has_error()` should be checked to determine if this method failed

```
$rc = $jnx->commit();  
  
if( $jnx->has_error ) {  
    $errmsg = NETCONF_util_errstr( $jnx );  
    print "ERROR committing config: $errmsg\n";  
    NETCONF_util_graceful_shutdown( $jnx );  
    exit 1;  
}
```

get_config

```
$rply = get_config( $args )
```

- This method is retrieve configuration from the target device. The commonly used arguments retrieving configurations are documented here
- \$args is a hash containing the following parameters:
 - `source` – options: ['running', 'candidate']. Most common is 'running' which will return the active configuration. The 'candidate' option returns the contents of the current configuration being modified
 - `filter` – A subset of the configuration can be specified by providing a 'filter' in the form of an XML configuration document (string form). This document must begin with the `<configuration>` element. See example on next page for using the `filter` option
- This NETCONF routine does not support returning inherit configuration values, i.e. the equivalent of "show | display inheritance".
- This NETCONF routine does not support retrieving the configuration in "text" (CLI) form
- Please see section "Customizing the NETCONF Driver" for details on making enhancements and customizations

get_config EXAMPLE

```
# -----  
# retrieve only the interface section of  
# the configuration file  
# -----  
  
$config = new JUNOS::EzEditXML;  
$config->interfaces;  
$XML_filter = $config->toString;  
  
<...snip code...>  
  
# -----  
# Retrieve the running configuration  
# -----  
  
$getargs{source} = 'running';  
  
if ($XML_filter ne '') {  
    $getargs{filter} = '<filter type="subtree">';  
    $getargs{filter} .= $XML_filter;  
    $getargs{filter} .= '</filter>';  
}  
  
$src = $jnx->get_config( %getargs );  
  
if( $jnx->has_error ) {  
    $errh = $jnx->get_first_error();  
    print "ERROR get-config: $errmsg\n";  
    print Dumper $errh;  
    exit 1;  
}
```

```
$clean_response = '';
```

```
cleanse_get_config( $src, \ $clean_response );
```

```
print ">>>>>>>>\n";  
print $clean_response;  
print "<<<<<<<<\n";
```

NOTE: The response from the `get_config` method (`$src`) includes the `<rpc>` header information. We are only interested in the `<configuration>` sections. The `cleanse_get_config()` utility routine is shown on the next slide and removes the unnecessary elements and attributes from the response

The `<filter>` definition is documented in the NETCONF guide, see Tag Element section on `<get-config>`

get_config EXAMPLE CONTINUED

```
sub cleanse_get_config {  
  
    my $_ = shift;  
    my $out_ref = shift;  
  
    # the return value from get_config looks like this:  
    # -----  
    # <rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" ...>  
    # <data>  
    # <configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" ...>  
    #     ... actual configuration response ...  
    # </configuration>  
    # </data>  
    # <rpc-reply>  
    # -----  
    #  
    # The we need to drop the <rpc-reply> and <data> elements. We  
    # also need to 'clean' the leading <configuration> block to remove  
    # the extra attributes. These attributes, in particular the  
    # namespace ('xmlns=...') can be troublesome for subsequent  
    # searching of the XML via XPath. So let's just return a clean  
    # XML response starting with <configuration> and ending with  
    # </configuration>.  
  
    m{<data>\r\n<confi.*>}g;    # find starting <configuration>  
    substr($_,0,pos $_) = '';   # erase from start including it  
    m{</configuration>}g;       # find ending </configuration>  
    substr($_,pos $_,-1) = '';  # erase from there to end of response  
  
    $$out_ref = '<configuration>' . $_;  
}
```

Once the response has been cleansed of extraneous elements and attributes it can easily be processed by XPath and other XML utilities.

ERROR HANDLING

<rpc-error>

- The <rpc-error> structure is returned from NETCONF. The XML structure is defined in the NETCONF guide and illustrated here

```
<rpc-error>
  <error-severity>error-severity</error-severity>
  <error-path>error-path</error-path>
  <error-message>error-message</error-message>
  <error-info>
    <bad-element>command-or-statement</bad-element>
  </error-info>
</rpc-error>
```

- The NETCONF Perl utility routines `get_first_error()` and `get_all_errors()` are used to retrieve these values in Perl hash form

ERROR HANDLING EXAMPLE

The following Perl script builds an invalid XML configuration filter, using an invalid element called “myint”. The response to the <get-config> request produces an <rpc-error>. The `get_first_error()` method is used to retrieve the error values in a Perl hash structure. The `Data::Dumper` utility displays the contents of this hash on the management system terminal.

Perl Script

```
use Data::Dumper;
<...snip...>
$config = new JUNOS::EzEditXML;
$config->interfaces->myint->name('xe-0/0/0');
$xml_filter = $config->toString;
print $xml_filter;

$src = $jnx->get_config( %getargs );

if( $jnx->has_error ) {
    $errh = $jnx->get_first_error();
    print "ERROR get-config: $errmsg\n";
    print Dumper $errh;
    exit 1;
}
```

Management System Output

```
<configuration>
  <interfaces>
    <myint>
      <name>xe-0/0/0</name>
    </myint>
  </interfaces>
</configuration>

ERROR get-config:
$VAR1 = {
    'error_message' => 'syntax error',
    'error_severity' => 'error',
    'bad_element' => 'myint'
};
```

LAB EXERCISES:

1. Create a Junos configuration script that performs basic device configuration
2. Create a Junos configuration script that creates a VLAN with a few interfaces
3. Create a Junos configuration script that removes one of the interfaces from the VLAN
4. Create a Junos configuration script that deactivates the VLAN
5. Create a Junos configuration script that activates the VLAN

LAB EXERCISES:

6. Create a Junos operational script that returns the results of “show ethernet-switching interfaces”
7. Create a Junos operational script that returns the result of “show version”
8. Create a Junos operational script the returns the result of “show chassis hardware detail”
9. Create a Junos operational script that returns the VLAN section of the configuration file



DEVELOP OPERATIONAL SCRIPTS

OPTION 1: USE TOOLKIT METHODS

STEP 1 – CAPTURE XML TEMPLATE

Login to the Junos device run the operational command and capture the results with the “display xml” option. Locate the first element after the <rpc-reply> tag. For example <route-information> follows if the command was “show route | display xml”. The corresponding XML command is the “get” version of the response, e.g. <get-route-information>

As of Junos 10.1 the CLI will support “| display xml rpc” which will provide the exact XML operational command definition and variables.

STEP 2 – LOOKUP XML COMMAND IN JUNIPER GUIDE

Most operational commands and their responses are documented in the “JUNOS XML Operational Reference” manual.

STEP 3 – USE NETCONF TOOLKIT API METHODS

The Juniper NETCONF Perl Toolkit includes a library of prebuilt methods to perform the operational commands. These methods will construct the necessary XML operational command structures and populate command variables. The following slides document how to locate these routines in the toolkit, understand the available parameters, and show code examples.

OPTION 2: DEVELOP XML OPERATIONAL COMMANDS

WHEN THE TOOLKIT DOES NOT PROVIDE

- There are times when you will find that the NETCONF toolkit does not provide a method for an operational command. This is commonly the case for much of the EX switching and SRX security features at present, but this situation is actively being resolved and full XML operational command support is expected in the near term
- These commands can be invoked via NETCONF, but you will need create the XML operational command and use the NETCONF device driver methods to send and receive the command & response
- This is a fairly easy and straightforward process. Examples will be provided on the following slides

USING TOOLKIT API METHODS

TOOLKIT API METHODS

LOCATION

- The API files can be found in the Perl installation directory under the Net directory. In a standard Cygwin environment using Perl v5.10 this would be:
`/lib/perl5/site_perl/5.10/Net/Netconf/Plugins/Plugin`
- Within the Plugin directory you will find directory for a specific release, e.g: 10.0R2
- Within the release directory you will find the API definitions files in the form of: `xxx_methods.pl`. For example the “show route” corresponds to the “get-route-information” API. This API can be found in the file `jroute_methods.pl`
- You can “grep” for the CLI command you are looking for and you will be able to find the API method

```
$ grep 'show route' *
jkernel_methods.pl:  ## Command: "show route forwarding-table"
jroute_methods.pl:   ## Command: "show route"
jroute_methods.pl:   ## Command: "show route instance"
jroute_methods.pl:   ## Command: "show route instance operational"
jroute_methods.pl:   ## Command: "show route summary"
jroute_methods.pl:   ## Command: "show route export"
jroute_methods.pl:   ## Command: "show route export vrf-target"
jroute_methods.pl:   ## Command: "show route export instance"
jroute_methods.pl:   ## Command: "show route flow validation"
```

USING TOOLKIT API

SHOW ROUTE EXAMPLE

- Using the “show route” command as an example, the next few slides illustrate the use of the toolkit API method
- Locate the API method in the xxx_methods.pl file. In this case <get-route-information> is located in jroute_methods.pl

```
get_route_information => {  
  logical_system => $STRING,  
  destination => $STRING,  
  all => $TOGGLE,  
  terse => $TOGGLE,  
  brief => $TOGGLE,  
  detail => $TOGGLE,  
  extensive => $TOGGLE,  
  best => $TOGGLE,  
  exact => $TOGGLE,  
  range => $TOGGLE,  
  table => $STRING,  
  label => $STRING,  
  ccc => $STRING,  
  inactive_prefix => $TOGGLE,  
  inactive_path => $TOGGLE,  
  active_path => $TOGGLE,  
  private => $TOGGLE,  
  damping => $STRING,  
  next_hop => $STRING,  
  source_gateway => $STRING,
```

```
  protocol => $STRING,  
  bgp => $TOGGLE,  
  dvmrp => $TOGGLE,  
  pim => $TOGGLE,  
  ripng => $TOGGLE,  
  rip => $TOGGLE,  
  msdp => $TOGGLE,  
  neighbor => $STRING,  
  bgp => $TOGGLE,  
  rip => $TOGGLE,  
  ripng => $TOGGLE,  
  dvmrp => $TOGGLE,  
  pim => $TOGGLE,  
  msdp => $TOGGLE,  
  peer => $STRING,  
  aspath_regex => $STRING,  
  no_community => $TOGGLE,  
  community_name => $STRING,  
  label_switched_path => $STRING,  
  hidden => $TOGGLE,  
},
```

SHOW ROUTE EXAMPLE

API METHOD VARIABLES

- Each API method defines a number of variable, `destination` is a STRING variable, for example. If we were to lookup the `<get-route-information>` description in the XML Operational Reference Manual, we would find the following:

`<destination>`—IP address and optional prefix length of destination.

- And running the “`show route 192.168/16 | display xml rpc`” command on the Junos CLI produces the XML operational command (does not run the command)

```
admin@SRX_lab1> show route 192.168/16 | display xml rpc
<rpc>
  <get-route-information>
    <destination>192.168.0.0/16</destination>
  </get-route-information>
</rpc>
```

- Based on the operational command and the variables desired, you can match these up to the toolkit API method defined variables

SHOW ROUTE EXAMPLE CONTINUED

The toolkit API method is invoked via the NETCONF object as shown below. The method name is the XML command element substituting dashes (-) with underscores (_)

```
$jnx = new Net::Netconf::Manager(  
    access => 'ssh',  
    login => $login,  
    password => $password,  
    hostname => $hostname );  
  
if(! $jnx ) {  
    print STDERR "Unable to connect to Junos device $hostname.\n";  
    exit 1;  
}  
  
print "Connection established: " . $jnx->get_session_id . "\n";  
  
$reply = $jnx->get_route_information( destination => '192.168/16' );  
  
if( $jnx->has_error ) {  
    $errh = $jnx->get_first_error();  
    print "ERROR get-config: $errmsg\n";  
    print Dumper $errh;  
    exit 1;  
}  
  
print "Return value.\n";  
print ">>>>>>>>\n";  
print $reply;  
print "<<<<<<<<\n";
```

The API method variables are passed as hash (key/value) pairs to the routine. Alternatively a hash structure can be created and passed in as a variable (e.g. as %oper args)

The result of the command is stored in the return variable (\$reply). The contents of this command are shown on the next slide

SHOW ROUTE EXAMPLE CONTINUED

The XML response in raw format looks like the following. Depending on your selection of XML processing tools, you may need to remove the XML namespace elements and attributes.

```
<rpc-reply
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:junos="http://xml.juniper.net/junos/10.1B3/junos">
  <route-information
xmlns="http://xml.juniper.net/junos/10.1B3/junos-
routing">
    <!-- keepalive -->
    <route-table>
      <table-name>inet.0</table-name>
      <destination-count>7</destination-count>
      <total-route-count>7</total-route-count>
      <active-route-count>7</active-route-count>
      <holddown-route-count>0</holddown-route-count>
      <hidden-route-count>0</hidden-route-count>
      <rt junos:style="brief">
        <rt-destination>192.168.2.0/24</rt-destination>
        <rt-entry>
          <active-tag>*</active-tag>
          <current-active/>
          <last-active/>
          <protocol-name>Direct</protocol-name>
          <preference>0</preference>
          <age junos:seconds="1952">00:32:32</age>
          <nh>
            <selected-next-hop/>
            <via>fe-0/0/5.0</via>
          </nh>
        </rt-entry>
      </rt>
    </route-table>
  </route-information>
</rpc-reply>
```

```
<rt junos:style="brief">
  <rt-destination>192.168.2.14/32</rt-destination>
  <rt-entry>
    <active-tag>*</active-tag>
    <current-active/>
    <last-active/>
    <protocol-name>Local</protocol-name>
    <preference>0</preference>
    <age junos:seconds="1961">00:32:41</age>
    <nh-type>Local</nh-type>
    <nh>
      <nh-local-interface>fe-0/0/5.0</nh-local-interface>
    </nh>
  </rt-entry>
</rt>
<rt junos:style="brief">
  <rt-destination>192.168.120.1/32</rt-destination>
  <rt-entry>
    <active-tag>*</active-tag>
    <current-active/>
    <last-active/>
    <protocol-name>Local</protocol-name>
    <preference>0</preference>
    <age junos:seconds="1961">00:32:41</age>
    <nh-type>Reject</nh-type>
  </rt-entry>
</rt>
</route-table>
</route-information>
</rpc-reply>
```

SHOW ROUTE EXAMPLE CONTINUED

The `cleanse_xmlns_attrs` routine is not part of the toolkit library, but is shown here to illustrate the technique for removing namespace and attributes.

```
$clean_rsp = '';
cleanse_xmlns_attrs( $src, \ $clean_rsp );

print "Return value.\n";
print ">>>>>>>>>\n";
print $clean_rsp;
print "<<<<<<<<<<\n";

sub cleanse_xmlns_attrs {
    my $_ = shift;
    my $out_ref = shift;

    s/^\<\/?rpc-reply.*>\/mg;          # remove <rpc-reply> tags
    s/^(<[\w-]+)(.*?)(\/?>)/\1\3/mg;  # remove attributes

    $$out_ref = $_;
}
```

The resulting output (`$clean_rsp`) is shown on the next slide.

SHOW ROUTE EXAMPLE CONTINUED

The XML response in “clean” format looks like the following. Variable XML and XPath processing utilities will have an easier time navigating the XML response structure.

```
<route-information>
  <!-- keepalive -->
  <route-table>
    <table-name>inet.0</table-name>
    <destination-count>7</destination-count>
    <total-route-count>7</total-route-count>
    <active-route-count>7</active-route-count>
    <holddown-route-count>0</holddown-route-count>
    <hidden-route-count>0</hidden-route-count>
    <rt>
      <rt-destination>192.168.2.0/24</rt-destination>
      <rt-entry>
        <active-tag>*</active-tag>
        <current-active/>
        <last-active/>
        <protocol-name>Direct</protocol-name>
        <preference>0</preference>
        <age>00:53:22</age>
        <nh>
          <selected-next-hop/>
          <via>fe-0/0/5.0</via>
        </nh>
      </rt-entry>
    </rt>
  </route-table>
</route-information>
```

```
<rt>
  <rt-destination>192.168.2.14/32</rt-destination>
  <rt-entry>
    <active-tag>*</active-tag>
    <current-active/>
    <last-active/>
    <protocol-name>Local</protocol-name>
    <preference>0</preference>
    <age>00:53:31</age>
    <nh-type>Local</nh-type>
    <nh>
      <nh-local-interface>fe-0/0/5.0</nh-local-interface>
    </nh>
  </rt-entry>
</rt>
<rt>
  <rt-destination>192.168.120.1/32</rt-destination>
  <rt-entry>
    <active-tag>*</active-tag>
    <current-active/>
    <last-active/>
    <protocol-name>Local</protocol-name>
    <preference>0</preference>
    <age>00:53:31</age>
    <nh-type>Reject</nh-type>
  </rt-entry>
</rt>
</route-table>
</route-information>
```

**DEVELOP XML OPERATIONAL
COMMANDS**

DEVELOP XML OPERATIONAL COMMANDS

NOT ENCOURAGED

- Developing XML operational commands should be an exception and not the rule. When you find the need to write an operational command, please contact Juniper Network and let them know there is a missing feature

SHOW ROUTE EXAMPLE

- Assume that the `<get-route-information>` toolkit API method was not available. How would one create the necessary XML command and perform the command/response action?
- Use `EzEditXML` to create the XML operational command
- Use the `opercmd_xml` method to perform the action on the Junos device. Note that this method is not part of the standard toolkit. Adding this method to the toolkit is covered in the “Customizing NETCONF Driver” section

SHOW ROUTE EXAMPLE

The Perl snippet to create the XML operational command and using the `opercmd_rpc` method is shown below. Note the XML operational tag is a parameter to the `EzEditXML` new method. Also note that the operational tag does not include the angle brackets (`<`, `>`).

```
$opercmd = new JUNOS::EzEditXML( 'get-route-information' );  
$opercmd->destination( '192.168/16' );  
  
$reply = $jnx->opercmd_rpc( opercmd_xml => $opercmd->toString );
```

As noted before, the `opercmd_rpc` is not part of the NETCONF Device methods. This method was added as a “custom” routine to support this capability. The process of customizing the NETCONF Driver methods is described in the next section.

CUSTOMIZING NETCONF DRIVER

CUSTOMIZING NETCONF DRIVER

NOT ENCOURAGED

- Developing custom NETCONF driver methods should be an exception and not the rule. When you find the need to write a driver method, please contact Juniper Network and let them know there is a missing feature

WHAT IS THE DRIVER?

- In the context of this section, the NETCONF driver refers to the device access methods such as “lock_config”, “edit_config”, “get_config”, “kill_session”.
- These routines can be found in the following location in the toolkit installation, on standard Cygwin environment: `/lib/perl5/site_perl/5.10/Net/Netconf/Plugins/Plugin`
File: `netconf.pm`

WHY CUSTOMIZE THE DRIVER

- You might find limitations in the existing driver. Features such as retrieving configuration with the equivalent of “display inheritance” or retrieve the configuration in text format are not supported in `<get-config>` but are available through another method.
- You might want to add the ability to perform XML operational commands (e.g. `opercmd_rpc`)

STEPS TO CUSTOMIZE NETCONF DRIVER

STEP-1: EDIT `netconf.pm`

- Locate the `netconf.pm` file in your toolkit installation. This can be found in the `Net/Netconf/Plugins/Plugin` directory of the toolkit installation

STEP-2: ADD method to `methods` hash structure

- Locate the `methods` hash structure. You will see the various driver methods defined, such as `lock_config`, `edit_config`, etc.
- Add your method and associated variable parameters

STEP-3: ADD method to `capabilities` hash structure

- Locate the `capabilities` hash structure and add your method to the first item list

STEP-4: WRITE your new driver method

- Write your method routine using the existing routines as a guide. A few example methods are shown on the following slides

netconf.pm

The methods hash structure contains the existing driver methods. Locate this structure in the netconf.pm file, as shown:

```
# Netconf APIs are generated using this hash
# This provides the parameters and attributes of the
# supported Netconf
# operations. The 'tag_name' value specifies the tag name
# to be used, if it is
# different than the hash key.
my %methods = (
    'get' => {
        filter => $DOM
    },
    'lock_config' => {
        'tag_name' => 'lock',
        'target' => $DOM_STRING,
        'target_url' => $URL_STRING
    },
    'unlock_config' => {
        'tag_name' => 'unlock',
        'target' => $DOM_STRING,
        'target_url' => $URL_STRING
    },
    'kill_session' => {
        'session_id' => $STRING
    },
    },
```

METHODS HASH STRUCTURE

The `opercmd_rpc` and `get_configuration` methods have been added to the methods hash structure, as shown:

```
my %methods = (  
    'get' => {  
        filter => $DOM  
    },  
    'lock_config' => {  
        'tag_name' => 'lock',  
        'target' => $DOM_STRING,  
        'target_url' => $URL_STRING  
    },  
    <...snip...>  
    'opercmd_rpc' => {  
        'opercmd_xml' => $DOM_STRING  
    },  
    'get_configuration' => {  
        'database' => $ATTRIBUTE,  
        'format' => $ATTRIBUTE,  
        'inherit' => $ATTRIBUTE,  
        'groups' => $ATTRIBUTE  
    }  
);
```

CAPABILITIES HASH STRUCTURE

The capabilities hash structure defines valid methods for the NETCONF driver. Add your custom methods to the end of the list, as shown:

```
# Capabilities supported by this plugin
my %capabilities = (
  'urn:ietf:params:xml:ns:netconf:base:1.0' =>
  [ 'get', 'get_config', 'edit_config', 'copy_config', 'delete_config',
    'lock_config', 'unlock_config', 'close_session', 'kill_session',
    'discard_changes', 'cli_rpc', 'load_config', 'opercmd_rpc',
    'get_configuration'
  ],
  'urn:ietf:params:xml:ns:netconf:capability:candidate:1.0' => [ 'commit' ],
  'urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0' => [],
  'urn:ietf:params:xml:ns:netconf:capability:validate:1.0' => [ 'validate' ],
  'urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file' =>
  [],
);
```

opercmd_rpc

The actual driver method code for `opercmd_rpc` is shown. It is a very simple routine that encapsulates the user-provided XML operational command between the `<rpc>` elements

```
sub opercmd_rpc
{
    my %args = @_;

    my $msg_id = &get_msg_id;
    my $request = '<rpc message-id=\"' . $msg_id . '\">';
    $request .= $args{opercmd_xml};
    $request .= '</rpc>';

    return $request;
}
```

get_configuration

The NETCONF driver method <get-config> does not support the ability to retrieve the configuration file in text format. Nor does it provide the ability to retrieve the configuration that includes the apply-group inheritance information. The Junos XML Protocol guide documents the <get-configuration> element tag which does support these capabilities.

It is a little known fact that all Junos XML Protocol element tags can be invoked in the context of NETCONF. That is, the internal Junos XML element processing engine handles the superset of NETCONF and Junos XML Protocol elements and attributes.

```
sub get_configuration
{
    my %args = @_;
    my $msg_id = &get_msg_id;
    my $request = '<rpc message-id=\"' . $msg_id . '\">';
    $request .= '<get-configuration';
    if( $args{database} ) { $request .= ' database=\"' . $args{database} . '\"'; }
    if( $args{format} ) { $request .= ' format=\"' . $args{format} . '\"'; }
    if( $args{groups} ) { $request .= ' groups=\"' . $args{groups} . '\"'; }
    if( $args{inherit} ) { $request .= ' inherit=\"' . $args{inherit} . '\"'; }
    $request .= '>';
    if( $args{configuration} ) { $request .= $args{configuration}; }
    else {
        $request .= '<configuration>';
        $request .= '</configuration>';
    }
    $request .= '</get-configuration>';
    $request .= '</rpc>';

    return $request;
}
```

FURTHER INFORMATION ON JUNOS AUTOMATION

JUNOS XML API DOCUMENTS AND PERL TOOLKIT

<https://www.juniper.net/support/xml/index.html>

JUNOS DAY-ONE BOOKS:

<http://www.juniper.net/us/en/community/junos/training-certification/day-one/>

JUNOS ON-BOX AUTOMATION SCRIPT LIBRARY:

<http://www.juniper.net/us/en/community/junos/script-automation/>

THANK YOU!



everywhere