NSO 4.4.2.3 Northbound APIs

First Published: May 17, 2010

Last Modified: September 1, 2017

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

CONTENTS

CHAPTER 1

Introduction

Cisco Network Service Orchestrator enabled by Tail-f (NSO) version 4.4.2.3 is an evolution of the Tail-f Network Control System (NCS). Tail-f was acquired by Cisco in 2014. The product has been enhanced and forms the base for Cisco NSO. Note that the terms 'ncs' and 'tail-f' are used extensively in file names, command-line command names, YANG models, application programming interfaces (API), etc. Throughout this document we will use NSO to mean the product, which consists of a number of modules and executable components. These executable components will be referred to by their command line name, e.g. **ncs**, **ncs-netsim**, **ncs_cli**, etc. **ncs** is also used to refer to the executable, the running daemon.

This chapter describes the various northbound programmatic APIs in NSO; NETCONF, REST, and SNMP. These APIs are used by external systems that need to communicate with NSO, such as portals, OSS, or BSS systems.

NSO has two northbound interfaces intended for human usage, the CLI and the WebUI. These interfaces are described in Chapter 2, *The NSO CLI* in *NSO 4.4.2.3 User Guide* and Chapter 11, *The Web User Interface* in *NSO 4.4.2.3 User Guide* respectively.

There are also programmatic Java APIs intended to be used by applications integrated with NSO itself. See ???? for more information about these APIs.

Integrating an External System with NSO

There are two APIs to choose from when an external system should communicate with NSO; NETCONF and REST. Which one to choose is mostly a subjective matter. REST may at first sight appear to be simpler to use, but is not as feature-rich as NETCONF. By using a NETCONF client library such as the open source Java library JNC or python library ncclient, the integration task is significantly reduced.

Both NETCONF and REST provides functions for manipulating the configuration (including creating services) and reading operational state from NSO. NETCONF provides more powerful filtering functions than REST.

NETCONF and SNMP can be used to receive alarms as notifications from NSO. NETCONF provides a reliable mechanism to receive notifications over SSH, whereas SNMP notifications are sent over UDP.

CHAPTER 2

The NCS NETCONF Server

Introduction

This chapter describes the north bound NETCONF implementation in NSO. As of this writing, the server supports the following specifications:

- RFC 4741 - NETCONF Configuration Protocol
- RFC 4742 - Using the NETCONF Configuration Protocol over Secure Shell (SSH)
- RFC 5277 - NETCONF Event Notifications
- RFC 5717 - Partial Lock Remote Procedure Call (RPC) for NETCONF
- RFC 6020 - YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)
- RFC 6021 - Common YANG Data Types
- RFC 6022 - YANG Module for NETCONF Monitoring
- RFC 6241 - Network Configuration Protocol (NETCONF)
- RFC 6242 - Using the NETCONF Configuration Protocol over Secure Shell (SSH)

- RFC 6243 - With-defaults capability for NETCONF
- RFC 6470 - NETCONF Base Notifications
- RFC 6536 - NETCONF Access Control Model
- RFC 6991 - Common YANG Data Types
- RFC 7895 - YANG Module Library
- RFC 7950 - The YANG 1.1 Data Modeling Language

Note For the <delete-config> operation specified in RFC 4741 / RFC 6241, only <url> with scheme "file" is supported for the <target> parameter - i.e. no data stores can be deleted. The concept of deleting a data store is not well defined, and at odds with the transaction-based configuration management of NSO. To delete the entire *contents* of a data store, with full transactional support, a <copy-config> with an empty <config/> element for the <source> parameter can be used.

NSO NETCONF north bound API can be used by arbitrary NETCONF clients. A simple Python based NETCONF client called `netconf-console` is shipped as source code in the distribution. See the section called “Using netconf-console” for details. Other NETCONF clients will work too, as long as they adhere to the NETCONF protocol. If you need a Java client, the open source client JNC can be used.

When integrating NCS into larger OSS/NMS environments, the NETCONF API is a good choice of integration point.

Capabilities

The NETCONF server in NCS supports the following capabilities in both NETCONF 1.0 (RFC 4741) and NETCONF 1.1 (RFC 6241).

`:writable-running`

This capability is always advertised.

`:candidate`

Not supported by NCS.

`:confirmed-commit`

Not supported by NCS.

`:rollback-on-error`

This capability allows the client to set the `<error-option>` parameter to `rollback-on-error`. The other permitted values are `stop-on-error` (default) and `continue-on-error`. Note that the meaning of the word "error" in this context is not defined in the specification. Instead, the meaning of this word must be defined by the data model. Also note that if `stop-on-error` or `continue-on-error` is triggered by the server, it means that some parts of the edit operation succeeded, and some parts didn't. The error `partial-operation` must be returned in this case. If some other error occurs (i.e. an error not covered by the meaning of "error" above), the server generates an appropriate error message, and the data store is unaffected by the operation.

The NSO server never allows partial configuration changes, since it might result in inconsistent configurations, and recovery from such a state can be very difficult for a client. This means that regardless of the value of the `<error-option>` parameter, NSO will always behave as if it had the value `rollback-on-error`. So in NSO, the meaning of the word "error" in `stop-on-error` and `continue-on-error`, is something which never can happen.

`:validate`

NSO supports both version 1.0 and 1.1 of this capability.

`:startup`
: Not supported by NCS.

`:url`
: The URL schemes supported are *`file`*, *`ftp`*, and *`sftp`* (SSH File Transfer Protocol).

There is no standard URL syntax for the *`sftp`* scheme, but NSO supports the syntax used by `curl`:

```
sftp://<user>:<password>@<host>/<path>
```

Note that user name and password must be given for *`sftp`* URLs.

`:xpath`
: The NETCONF server supports XPath according to the W3C XPath 1.0 specification (http://www.w3.org/TR/xpath).

The following list of optional standard capabilities are also supported:

`:notification`
: NSO implements the `urn:ietf:params:netconf:capability:notification:1.0` capability, including support for the optional replay feature.

See the section called “Notification Capability” for details.

`:with-defaults`
: NSO implements the `urn:ietf:params:netconf:capability:with-defaults:1.0` capability, which is used by the server to inform the client how default values are handled by the server, and by the client to control whether defaults values should be generated to replies or not.

`:yang-library`
: NSO implements the `urn:ietf:params:netconf:capability:yang-library:1.0` capability, which informs the client that server implements the YANG module library RFC 7895, and informs the client about the current `module-set-id`.

In addition to the standard capabilities NCS also includes the following optional, non-standard capabilities.

`actions`
: See the section called “Actions Capability” for details.

`transactions`
: See the section called “Transactions Capability” for details.

`inactive`
: See the section called “Inactive Capability” for details.

The server reports each data model namespace it has loaded as separate capabilities, according to the YANG specification.

Advertising Capabilities and YANG Modules

All enabled NETCONF capabilities are advertised in the hello message that the server sends to the client.

A YANG module is supported by the NETCONF server if it's fxs file is found in NCS's loadPath, and if the fxs files is exported to NETCONF.

The following YANG modules are built-in, which means that their fxs files must not be present in the loadPath:

- `ietf-netconf`

- `ietf-netconf-with-defaults`
- `ietf-yang-library`
- `ietf-yang-types`
- `ietf-inet-types`

All built-in modules are always supported by the server.

All YANG version 1 modules supported by the server are advertised in the hello message, according to the rules defined in RFC 6020.

All YANG version 1 and version 1.1 modules supported by the server are advertised in the `module` list defined in `ietf-yang-library`.

If a YANG module (any version) is supported by the server, and its .yang or .yin file is found in the fxs file or in the loadPath, then the module is also advertised in the `schema` list defined in `ietf-netconf-monitoring`, made available for download with the RPC operation `get-schema`, and if RESTCONF is enabled, also advertised in the `schema` leaf in `ietf-yang-library`. See the section called “Monitoring of the NETCONF Server”.

NETCONF Transport Protocols

The NETCONF server natively supports the mandatory SSH transport, i.e., SSH is supported without the need for an external SSH daemon (such as sshd). It also supports integration with OpenSSH.

Using OpenSSH

NSO is delivered with a program **netconf-subsys** which is an OpenSSH *subsystem* program. It is invoked by the OpenSSH daemon after successful authentication. It functions as a relay between the ssh daemon and NSO; it reads data from the ssh daemon from standard input, and writes the data to NSO over a loopback socket, and vice versa. This program is delivered as source code in `$NCS_DIR/src/ncs/netconf/netconf-subsys.c`. It can be modified to fit the needs of the application. For example, it could be modified to read the group names for a user from an external LDAP server.

When using OpenSSH, the users are authenticated by OpenSSH, i.e. the user names are not stored in NSO. To use OpenSSH, compile the **netconf-subsys** program, and put the executable in e.g. `/usr/local/bin`. Then add the following line to the ssh daemon's config file, `sshd_config`:

```
Subsystem     netconf   /usr/local/bin/netconf-subsys
```

The connection from **netconf-subsys** to NSO can be arranged in one of two different ways:

1. Make sure NSO is configured to listen to TCP traffic on localhost, port 2023, and disable SSH in `ncs.conf` (see ncs.conf(5) in *NSO 4.4.2.3 Manual Pages*). (Re)start sshd and NSO. Or:
2. Compile **netconf-subsys** to use a connection to the IPC port instead of the NETCONF TCP transport (see the `netconf-subsys.c` source for details), and disable both TCP and SSH in `ncs.conf`. (Re)start sshd and NSO.

   This method may be preferable, since it makes it possible to use the IPC Access Check (see the section called “Restricting access to the IPC port” in *NSO 4.4.2.3 Administration Guide*) to restrict the unauthenticated access to NSO that is needed by **netconf-subsys**.

By default the **netconf-subsys** program sends the names of the UNIX groups the authenticated user belongs to. To test this, make sure that NSO is configured to give access to the group(s) the user belongs to. Easiest for test is to give access to all groups.

Configuration of the NETCONF Server

NCS itself is configured through a configuration file called `ncs.conf`. For a description of the parameters in this file, please see the ncs.conf(5) in *NSO 4.4.2.3 Manual Pages* man page.

Error Handling

When NSO processes `<get>`, `<get-config>`, and `<copy-config>` requests, the resulting data set can be very large. To avoid buffering huge amounts of data, NSO streams the reply to the client as it traverses the data tree and calls data provider functions to retrieve the data.

If a data provider fails to return the data it is supposed to return, NSO can take one of two actions. Either it simply closes the NETCONF transport (default), or it can reply with an *inline rpc error* and continue to process the next data element. This behavior can be controlled with the `/ncs-config/netconf/rpc-errors` configuration parameter (see ncs.conf(5) in *NSO 4.4.2.3 Manual Pages*).

An inline error is always generated as a child element to the parent of the faulty element. For example, if an error occurs when retrieving the leaf element "mac-address" of an "interface" the error might be:

```
<interface>
  <name>atm1</name>
  <rpc-error xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <error-type>application</error-type>
    <error-tag>operation-failed</error-tag>
    <error-severity>error</error-severity>
    <error-message xml:lang="en">Failed to talk to hardware</error-message>
    <error-info>
      <bad-element>mac-address</bad-element>
    </error-info>
  </rpc-error>
  ...
</interface>
```

If a `get_next` call fails in the processing of a list, a reply might look like this:

```
<interface>
  <!-- successfully retrieved list entry -->
  <name>eth0</name>
  <mtu>1500</mtu>
  <!-- more leafs here -->
</interface>
<rpc-error xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <error-type>application</error-type>
  <error-tag>operation-failed</error-tag>
  <error-severity>error</error-severity>
  <error-message xml:lang="en">Failed to talk to hardware</error-message>
  <error-info>
    <bad-element>interface</bad-element>
  </error-info>
</rpc-error>
```

Monitoring of the NETCONF Server

RFC 6022 - YANG Module for NETCONF Monitoring defines a YANG module, `ietf-netconf-monitoring`, for monitoring of the NETCONF server. It contains statistics objects such as number of RPCs received, status objects such as user sessions, and an operation to retrieve data models from the NETCONF server.

This data model defines a new RPC operation, `get-schema`, which is used to retrieve YANG modules from the NETCONF server. NSO will report the YANG modules for all fxs files that are reported as capabilities, and for which the corresponding YANG or YIN file is stored in the fxs file or found in the loadPath. If a file is found in the loadPath, it has priority over a file stored in the fxs file. Note that by default, the module and its submodules are stored in the fxs file by the compiler.

If the YANG (or YIN files) are copied into the loadPath, they can be stored as is or compressed with gzip. The filename extension MUST be ".yang", ".yin", ".yang.gz", or ".yin.gz".

Also available is a Tail-f specific data model, `tailf-netconf-monitoring`, which augments `ietf-netconf-monitoring` with additional data about files available for usage with the `<copy-config>` command with a *file* `<url>` source or target. `/ncs-config/netconf-north-bound/capabilities/url/enabled` and `/ncs-config/netconf-north-bound/capabilities/url/file/enabled` must both be set to true. If rollbacks are enabled, those files are listed as well, and they can be loaded using `<copy-config>`.

This data model also adds data about which notification streams are present in the system, and data about sessions that subscribe to the streams.

Notification Capability

This section describes how NETCONF notifications are implemented within NSO, and how the applications generates these events.

Central to NETCONF notifications is the concept of a *stream*. The stream serves two purposes. It works like a high-level filtering mechanism for the client. For example, if the client subscribes to notifications on the `security` stream, it can expect to get security related notifications only. Second, each stream may have its own log mechanism. For example by keeping all debug notifications in a `debug` stream, they can be logged separately from the `security` stream.

Built-in Notification Streams

NCS has built-in support for the well-known stream `NETCONF`, defined in RFC 5277. NCS supports the notifications defined in RFC 6470 - NETCONF Base Notifications on this stream. If the application needs to send any additional notifications on this stream, it can do so.

NCS can be configured to listen to NETCONF notifications from devices, and send those notifications to northbound NETCONF clients. The stream `device-notifications` is used for this purpose. In order to enable this, the stream `device-notifications` must be configured in `ncs.conf`, and additionally, subscriptions must be created in `/ncs:devices/device/netconf-notifications`.

Defining Notification Streams

It is up to the application to define which streams it supports. In NCS, this is done in `ncs.conf` (see ncs.conf(5) in *NSO 4.4.2.3 Manual Pages*). Each stream must be listed, and whether it supports replay or not. The following example enables the built-in stream `device-notifications` with replay support, and an additional, application-specific stream `debug` without replay support:

```
<notifications>
  <event-streams>
    <stream>
      <name>device-notifications</name>
      <description>Notifications received from devices</description>
      <replay-support>true</replaySupport>
      <builtin-replays-store>
```

```
        <enabled>true</enabled>
        <dir>/var/log</dir>
        <max-size>S10M</maxSize>
        <max-files>50</maxFiles>
      </builtin-replay-store>
    </stream>
    <stream>
      <name>debug</name>
      <description>Debug notifications</description>
      <replay-support>false</replay-support>
    </stream>
  </event-streams>
</notifications>
```

The well-known stream NETCONF does not have to be listed, but if it isn't listed, it will not support replay.

Automatic Replay

NSO has builtin support for logging of notifications, i.e., if replay support has been enabled for a stream, NSO automatically stores all notifications on disk ready to be replayed should a NETCONF client ask for logged notifications. In the `ncs.conf` fragment above the security stream has been setup to use the builtin notification log/replay store. The replay store uses a set of wrapping log files on disk (of a certain number and size) to store the security stream notifications.

The reason for using a wrap log is to improve replay performance whenever a NETCONF client asks for notifications in a certain time range. Any problems with log files not being properly closed due to hard power failures etc. is also kept to a minimum, i.e., automatically taken care of by NSO.

Using netconf-console

The `netconf-console` program is a simple NETCONF client. It is delivered as Python source code and can be used as-is or modified.

When NSO has been started, we can use `netconf-console` to query the configuration of the NETCONF Access Control groups:

```
$ netconf-console --get-config -x /nacm/groups
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <nacm xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-acm">
      <groups>
        <group>
          <name>admin</name>
          <user-name>admin</user-name>
          <user-name>private</user-name>
        </group>
        <group>
          <name>oper</name>
          <user-name>oper</user-name>
          <user-name>public</user-name>
        </group>
      </groups>
    </nacm>
  </data>
</rpc-reply>
```

With the `-x` flag an XPath expression can be specified, in order to retrieve only data matching that expression. This is a very convenient way to extract portions of the configuration from the shell or from shell scripts.

Actions Capability

Overview

This capability introduces a new RPC method which is used to invoke actions defined in the data model. When an action is invoked, the instance on which the action is invoked is explicitly identified by an hierarchy of configuration or state data.

Here is a simple example that invokes the action `sync-from` on the device `ce1`. It uses the **netconf-console** command:

```
$ cat ./sync-from-ce1.xml
<action xmlns="http://tail-f.com/ns/netconf/actions/1.0">
  <data>
    <devices xmlns="http://tail-f.com/ns/ncs">
      <device>
        <name>ce1</name>
        <sync-from/>
      </device>
    </devices>
  </data>
</action>
$ netconf-console --rpc sync-from-ce1.xml
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
  <data>
    <devices xmlns="http://tail-f.com/ns/ncs">
      <device>
        <name>ce1</name>
        <sync-from>
          <result>true</result>
        </sync-from>
      </device>
    </devices>
  </data>
</rpc-reply>
```

Capability Identifier

The actions capability is identified by the following capability string:

```
  http://tail-f.com/ns/netconf/actions/1.0
```

Transactions Capability

Overview

This capability introduces four new rpc methods which are used to control a two-phase commit transaction on the NETCONF server. The normal <edit-config> operation is used to write data in the transaction, but the modifications are not applied until an explicit <commit-transaction> is sent.

This capability is formally defined in the YANG module "tailf-netconf-transactions".

A typical sequence of operations looks like this:

```
               C                            S
               |                            |
               |  capability exchange      |
               |-------------------------->|
               |<------------------------->|
               |                           |
               |   <start-transaction>     |
               |-------------------------->|
               |<--------------------------|
               |         <ok/>             |
               |                           |
               |     <edit-config>         |
               |-------------------------->|
               |<--------------------------|
               |         <ok/>             |
               |                           |
               |  <prepare-transaction>    |
               |-------------------------->|
               |<--------------------------|
               |         <ok/>             |
               |                           |
               |   <commit-transaction>    |
               |-------------------------->|
               |<--------------------------|
               |         <ok/>             |
               |                           |
```

Dependencies

None.

Capability Identifier

The transactions capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/transactions/1.0
```

New Operation: <start-transaction>

Description

Starts a transaction towards a configuration datastore. There can be a single ongoing transaction per session at any time.

When a transaction has been started, the client can send any NETCONF operation, but any <edit-config> or <copy-config> operation sent from the client MUST specify the same <target> as the <start-transaction>, and any <get-config> MUST specify the same <source> as <start-transaction>.

If the server receives an <edit-config> or <copy-config> with another <target>, or a <get-config> with another <source>, an error MUST be returned with an <error-tag> set to "invalid-value".

The modifications sent in the <edit-config> operations are not immediately applied to the configuration datastore. Instead they are kept in the transaction state of the server. The transaction state is only applied when a <commit-transaction> is received.

The client sends a <prepare-transaction> when all modifications have been sent.

Parameters

target:	Name of the configuration datastore towards which the transaction is started.

with-inactive: If this parameter is given, the transaction will handle the "inactive" and "active" attributes. If given, it MUST also be given in the <edit-config> and <get-config> invocations in the transaction.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is an ongoing transaction for this session already, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <start-transaction xmlns="http://tail-f.com/ns/netconf/transactions/1.0">
    <target>
     <running/>
    </target>
  </start-transaction>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

New Operation: <prepare-transaction>

Description

Prepares the transaction state for commit. The server may reject the prepare request for any reason, for example due to lack of resources or if the combined changes would result in an invalid configuration datastore.

After a successful <prepare-transaction>, the next transaction related rpc operation must be <commit-transaction> or <abort-transaction>. Note that an <edit-config> cannot be sent before the transaction is either committed or aborted.

Care must be taken by the server to make sure that if <prepare-transaction> succeeds then the <commit-transaction> SHOULD not fail, since this might result in an inconsistent distributed state. Thus, <prepare-transaction> should allocate any resources needed to make sure the <commit-transaction> will succeed.

Parameters

None.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is no ongoing transaction in this session, or if the ongoing transaction already has been prepared, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <prepare-transaction
     xmlns="http://tail-f.com/ns/netconf/transactions/1.0"/>
</rpc>

<rpc-reply message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

New Operation: <commit-transaction>

Description

Applies the changes made in the transaction to the configuration datatore. The transaction is closed after a <commit-transaction>.

Parameters

None.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is no ongoing transaction in this session, or if the ongoing transaction already has not been prepared, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit-transaction
     xmlns="http://tail-f.com/ns/netconf/transactions/1.0"/>
</rpc>

<rpc-reply message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

New Operation: <abort-transaction>

Description

Aborts the ongoing transaction, and all pending changes are discarded. <abort-transaction> can be given at any time during an ongoing transaction.

Parameters

None.

Positive Response

If the device was able to satisfy the request, an <rpc-reply> is sent that contains an <ok> element.

Negative Response

An <rpc-error> element is included in the <rpc-reply> if the request cannot be completed for any reason.

If there is no ongoing transaction in this session, an error MUST be returned with <error-app-tag> set to "bad-state".

Example

```
<rpc message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <abort-transaction
     xmlns="http://tail-f.com/ns/netconf/transactions/1.0"/>
</rpc>

<rpc-reply message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

Modifications to Existing Operations

The <edit-config> operation is modified so that if it is received during an ongoing transaction, the modifications are not immediately applied to the configuration target. Instead they are kept in the transaction state of the server. The transaction state is only applied when a <commit-transaction> is received.

Note that it doesn't matter if the <test-option> is 'set' or 'test-then-set' in the <edit-config>, since nothing is actually set when the <edit-config> is received.

XML Schema

This XML Schema defines the new transaction rpcs.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://tail-f.com/ns/netconf/transactions/1.0"
    xmlns="http://tail-f.com/ns/netconf/transactions/1.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:netconf="urn:ietf:params:xml:ns:netconf:base:1.0"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified"
    xml:lang="en">

  <!-- Type for <target> element -->
  <xs:complexType name="TargetType">
    <xs:choice>
      <xs:element name="running"/>
      <xs:element name="startup"/>
      <xs:element name="candidate"/>
    </xs:choice>
  </xs:complexType>
```

```
    <!--  <start-transaction> operation -->
    <xs:complexType name="StartTransactionType">
      <xs:complexContent>
        <xs:extension base="netconf:rpcOperationType">
          <xs:sequence>
            <xs:element name="target" type="TargetType"/>
            <xs:element name="with-inactive" minOccurs="0"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>

    <xs:element name="start-transaction" type="StartTransactionType"
                substitutionGroup="netconf:rpcOperation"/>

    <xs:element name="prepare-transaction"
                substitutionGroup="netconf:rpcOperation"/>

    <xs:element name="commit-transaction"
                substitutionGroup="netconf:rpcOperation"/>

    <xs:element name="abort-transaction"
                substitutionGroup="netconf:rpcOperation"/>

  </xs:schema>
```

Inactive Capability

Overview

This capability is used by the NETCONF server to indicate that it supports marking nodes as being inactive. A node that is marked as inactive exists in the data store, but is not used by the server. Any node can be marked as inactive.

In order to not confuse clients that do not understand this attribute, the client has to instruct the server to display and handle the inactive nodes. An inactive node is marked with an "inactive" XML attribute, and in order to make it active, the "active" XML atribute is used.

This capability is formally defined in the YANG module "tailf-netconf-inactive".

Dependencies

None.

Capability Identifier

The inactive capability is identified by the following capability string:

```
http://tail-f.com/ns/netconf/inactive/1.0
```

New Operations

None.

Modifications to Existing Operations

A new parameter, <with-inactive>, is added to the <get>, <get-config>, <edit-config>, <copy-config>, and <start-transaction> operations.

The <with-inactive> element is defined in the http://tail-f.com/ns/netconf/inactive/1.0 namespace, and takes no value.

If this parameter is present in <get>, <get-config>, or <copy-config>, the NETCONF server will mark inactive nodes with the "inactive" attribute.

If this parameter is present in <edit-config> or <copy-config>, the NETCONF server will treat inactive nodes as existing, so that an attempt to create a node which is inactive will fail, and an attempt to delete a node which is inactive will succeed. Further, the NETCONF server accepts the "inactive" and "active" attributes in the data hierarchy, in order to make nodes inactive or active, respectively.

If the parameter is present in <start-transaction>, it MUST also be present in any <edit-config>, <copy-config>, <get>, or <get-config> operations within the transaction. If it is not present in <start-transaction>, it MUST NOT be present in any <edit-config> operation within the transaction.

The "inactive" and "active" attributes are defined in the http://tail-f.com/ns/netconf/inactive/1.0 namespace. The "inactive" attribute's value is the string "inactive", and the "active" attribute's value is the string "active".

Example

This request creates an inactive interface:

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <with-inactive
       xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
    <config>
      <top xmlns="http://example.com/schema/1.2/config">
        <interface inactive="inactive">
          <name>Ethernet0/0</name>
          <mtu>1500</mtu>
        </interface>
      </top>
    </config>
  </edit-config>
</rpc>

<rpc-reply message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

This request shows the inactive interface:

```
<rpc message-id="102"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <with-inactive
       xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
  </get-config>
</rpc>

<rpc-reply message-id="102"
```

```
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <top xmlns="http://example.com/schema/1.2/config">
      <interface inactive="inactive">
        <name>Ethernet0/0</name>
        <mtu>1500</mtu>
      </interface>
    </top>
  </data>
</rpc-reply>
```

This request shows that inactive data is not returned unless the client asks for it:

```
<rpc message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>

<rpc-reply message-id="103"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
  </data>
</rpc-reply>
```

This request activates the interface:

This request creates an inactive interface:

```
<rpc message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <with-inactive
       xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
    <config>
      <top xmlns="http://example.com/schema/1.2/config">
        <interface active="active">
          <name>Ethernet0/0</name>
        </interface>
      </top>
    </config>
  </edit-config>
</rpc>

<rpc-reply message-id="104"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <ok/>
</rpc-reply>
```

NETCONF extensions in NCS

The YANG module `tailf-netconf-ncs` augments some NETCONF operations with additional parameters to control the commit-behavior in NCS over NETCONF. See that YANG module for all details. In this section the options are summarized.

The following additional input parameters are available:

- `no-revision-drop` - This flag forces NCS to never silently drop any data set operations towards a device.
- `no-overwrite` - The flag means that NCS will check that the data that should be modified has not changed on the device compared to NCS's view of the data.
- `no-networking` - This flag forces NCS to not send any data to the devices.
- `no-out-of-sync-check` - This flag means that NCS will continue with the transaction even if it detects that a device's configuration is out of sync.
- `commit-queue/tag` - User defined opaque tag which is present in notifications for this queue item.
- `commit-queue/async` - Commit through the commit queue but do not wait for a reply.
- `commit-queue/sync/timeout` - Commit through the commit-queue and wait for completion; wait max the specified number of seconds.
- `commit-queue/sync/infinity` - Commit through the commit-queue and wait for completion.
- `commit-queue/bypass` - Do not commit through the commit queue.
- `commit-queue/block-others` - The resulting queue item will block subsequent queue items, which use any of the devices in this queue item, from being queued.

The following NETCONF operations are augmented with these optional input parameters:

- `commit`
- `edit-config`
- `copy-config`
- `prepare-transaction`

The operation `prepare-transaction` is also augmented with an optional parameter `dryrun`, which can be used to run all service code, but not actually commit anything to the datastore or to the devices. `dryrun` takes an optional parameter `outformat`, which can be used to select in which format the result is returned. It defaults to `xml`.

The Query API

The Query API consists of a number of RPC operations to start queries, fetch chunks of the result from a query, restart a query, and stop a query.

In the installed release there are two YANG files named `tailf-netconf-query.yang` and `tailf-common-query.yang` that defines these operations. An easy way to find the files is to run the following command from the top directory of release installation:

```
$ find . -name tailf-netconf-query.yang
```

The API consists of the following operations:

- `start-query`: Start a query and return a query handle.
- `fetch-query-result`: Use a query handle to repeatedly fetch chunks of the result.
- `reset-query`: (Re)set where the next fetched result will begin from.
- `stop-query`: Stop (and close) the query.

In the following examples, the following data model is used:

```
container x {
  list host {
    key number;
```

```
      leaf number {
        type int32;
      }
      leaf enabled {
        type boolean;
      }
      leaf name {
        type string;
      }
      leaf address {
        type inet:ip-address;
      }
    }
  }
```

Here is an example of a `start-query` operation:

```
<start-query xmlns="http://tail-f.com/ns/netconf/query">
  <foreach>
    /x/host[enabled = 'true']
  </foreach>
  <select>
    <label>Host name</label>
    <expression>name</expression>
    <result-type>string</result-type>
  </select>
  <select>
    <expression>address</expression>
    <result-type>string</result-type>
  </select>
  <sort-by>name</sort-by>
  <limit>100</limit>
  <offset>1</offset>
</start-query>
```

An informal interpretation of this query is:

For each `/x/host` where `enabled` is true, select its `name`, and `address`, and return the result sorted by `name`, in chunks of 100 results at the time.

Let us discuss the various pieces of this request.

The actual XPath query to run is specified by the `foreach` element. In the example below will search for all `/x/host` nodes that has the `enabled` node set to `true`:

```
<foreach>
  /x/host[enabled = 'true']
</foreach>
```

Now we need to define what we want to have returned from the node set by using one or more `select` sections. What to actually return is defined by the XPath `expression`.

We must also choose how the result should be represented. Basically, it can be the actual value or the path leading to the value. This is specified per select chunk The possible result-types are: `string` , `path` , `leaf-value` and `inline`.

The difference between `string` and `leaf-value` is somewhat subtle. In the case of `string` the result will be processed by the XPath function `string()` (which if the result is a node-set will concatenate all the values). The `leaf-value` will return the value of the first node in the result. As long as the result is a leaf node, `string` and `leaf-value` will return the same result. In the example above, we are using `string` as shown below. At least one `result-type` must be specified.

The result-type `inline` makes it possible to return the full sub-tree of data in XML format. The data will be enclosed with a tag: `data`.

Finally we can specify an optional `label` for a convenient way of labeling the returned data. In the example we have the following:

```
<select>
  <label>Host name</label>
  <expression>name</expression>
  <result-type>string</result-type>
</select>
<select>
  <expression>address</expression>
  <result-type>string</result-type>
</select>
```

The returned result can be sorted. This is expressed as XPath expressions, which in most cases are very simple and refers to the found node set. In this example we sort the result by the content of the `name` node:

```
<sort-by>name</sort-by>
```

To limit the max amount of results in each chunk that `fetch-query-result` will return we can set the `limit` element. The default is to get all results in one chunk.

```
<limit>100</limit>
```

With the `offset` element we can specify at which node we should start to receive the result. The default is 1, i.e., the first node in the resulting node-set.

```
<offset>1</offset>
```

Now, if we continue by putting the operation above in a file `query.xml` we can send a request, using the command **netconf-console**, like this:

```
$ netconf-console --rpc query.xml
```

The result would look something like this:

```
<start-query-result>
  <query-handle>12345</query-handle>
</start-query-result>
```

The query handle (in this example "12345") must be used in all subsequent calls. To retrieve the result, we can now send:

```
<fetch-query-result xmlns="http://tail-f.com/ns/netconf/query">
  <query-handle>12345</query-handle>
</fetch-query-result>
```

Which will result in something like the following:

```
<query-result xmlns="http://tail-f.com/ns/netconf/query">
  <result>
    <select>
      <label>Host name</label>
      <value>One</value>
    </select>
    <select>
      <value>10.0.0.1</value>
    </select>
  </result>
  <result>
```

```
    <select>
      <label>Host name</label>
      <value>Three</value>
    </select>
    <select>
      <value>10.0.0.1</value>
    </select>
  </result>
</query-result>
```

If we try to get more data with the `fetch-query-result` we might get more `result` entries in return until no more data exists and we get an empty query result back:

```
<query-result xmlns="http://tail-f.com/ns/netconf/query">
</query-result>
```

If we want to go back in the "stream" of received data chunks and have them repeated, we can do that with the `reset-query` operation. In the example below we ask to get results from the 42:nd result entry:

```
<reset-query xmlns=\"http://tail-f.com/ns/netconf/query\">
  <query-handle>12345</query-handle>
  <offset>42</offset>
</reset-query>
```

Finally, when we are done we stop the query:

```
<stop-query xmlns="http://tail-f.com/ns/netconf/query">
  <query-handle>12345</query-handle>
</stop-query>
```

Meta-data in Attributes

NSO supports three pieces of meta-data data nodes: tags, annotations, and inactive.

An annotation is a string which acts a comment. Any data node present in the configuration can get an annotation. An annotation does not affect the underlying configuration, but can be set by a user to comment what the configuration does.

An annotation is encoded as an XML attribute 'annotation' on any data node. To remove an annotation, set the 'annotation' attribute to an empty string.

Any configuration data node can have a set of tags. Tags are set by the user for data organization and filtering purposes. A tag does not affect the underlying configuration.

All tags on a data node are encoded as a space separated string in an XML attribute 'tags'. To remove all tags, set the 'tags' attribute to an empty string.

Annotation, tags, and inactive attributes can be present in `<edit-config>`, `<copy-config>`, `<get-config>`, and `<get>`. For example:

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <interfaces xmlns="http://example.com/ns/if">
        <interface annotation="this is the management interface"
                   tags=" important ethernet ">
          <name>eth0</name>
          ...
        </interface>
      </interfaces>
    </config>
  </edit-config>
</rpc>
```

Namespace for Additional Error Information

NSO adds an additional namespace which is used to define elements which are included in the `<error-info>` element. This namespace also describes which `<error-app-tag/>` elements the server might generate, as part of an `<rpc-error/>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://tail-f.com/ns/netconf/params/1.1"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xml:lang="en">

  <xs:annotation>
    <xs:documentation>
      Tail-f's namespace for additional error information.
      This namespace is used to define elements which are included
      in the 'error-info' element.

      The following are the app-tags used by the NETCONF agent:

        o  not-writable

          Means that an edit-config or copy-config operation was
          attempted on an element which is read-only
          (i.e. non-configuration data).

        o  missing-element-in-choice

          Like the standard error missing-element, but generated when
          one of a set of elements in a choice is missing.

        o  pending-changes

          Means that a lock operation was attempted on the candidate
          database, and the candidate database has uncommitted
          changes. This is not allowed according to the protocol
          specification.

        o  url-open-failed

          Means that the URL given was correct, but that it could not
          be opened. This can e.g. be due to a missing local file, or
          bad ftp credentials. An error message string is provided in
          the &lt;error-message&gt; element.

        o  url-write-failed

          Means that the URL given was opened, but write failed. This
          could e.g. be due to lack of disk space. An error message
          string is provided in the &lt;error-message&gt; element.

        o  bad-state

          Means that an rpc is received when the session is in a state
          which don't accept this rpc.  An example is
          &lt;prepare-transaction&gt; before &lt;start-transaction&gt;

    </xs:documentation>
  </xs:annotation>

  <xs:element name="bad-keyref">
    <xs:annotation>
      <xs:documentation>
        This element will be present in the 'error-info' container when
        'error-app-tag' is "instance-required".
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="bad-element" type="xs:string">
          <xs:annotation>
            <xs:documentation>
              Contains an absolute XPath expression pointing to the element
              which value refers to a non-existing instance.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="missing-element" type="xs:string">
          <xs:annotation>
            <xs:documentation>
              Contains an absolute XPath expression pointing to the missing
              element referred to by 'bad-element'.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="bad-instance-count">
    <xs:annotation>
      <xs:documentation>
        This element will be present in the 'error-info' container when
        'error-app-tag' is "too-few-elements" or "too-many-elements".
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="bad-element" type="xs:string">
          <xs:annotation>
            <xs:documentation>
              Contains an absolute XPath expression pointing to an
              element which exists in too few or too many instances.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="instances" type="xs:unsignedInt">
          <xs:annotation>
            <xs:documentation>
              Contains the number of existing instances of the element
              referd to by 'bad-element'.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:choice>
          <xs:element name="min-instances" type="xs:unsignedInt">
            <xs:annotation>
```

```
              <xs:documentation>
                Contains the minimum number of instances that must
                exist in order for the configuration to be consistent.
                This element is present only if 'app-tag' is
                'too-few-elems'.
              </xs:documentation>
            </xs:annotation>
          </xs:element>
          <xs:element name="max-instances" type="xs:unsignedInt">
            <xs:annotation>
              <xs:documentation>
                Contains the maximum number of instances that can
                exist in order for the configuration to be consistent.
                This element is present only if 'app-tag' is
                'too-many-elems'.
              </xs:documentation>
            </xs:annotation>
          </xs:element>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:attribute name="annotation" type="xs:string">
    <xs:annotation>
      <xs:documentation>
        This attribute can be present on any configuration data node.  It
        acts as a comment for the node.  The annotation does not affect the
        underlying configuration data.
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>

  <xs:attribute name="tags" type="xs:string">
    <xs:annotation>
      <xs:documentation>
        This attribute can be present on any configuration data node.  It
        is a space separated string of tags for the node.  The tags of a
        node does not affect the underlying configuration data, but can
        be used by a user for data organization, and data filtering.
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>

</xs:schema>
```

CHAPTER 3

The REST API

Introduction

This chapter describes a RESTful API over HTTP for accessing data defined in YANG. It tries to follow the RESTCONF Internet Draft [draft-ietf-netconf-restconf] but since it predates the creation of RESTCONF, a number of differences exists. Whenever such a difference occur, it is clearly stated.

The RESTCONF protocol operates in the configuration datastores defined in NETCONF. It defines a set of Create, Retrieve, Update, Delete (CRUD) operations that can be used to access these datastores. The YANG language defines the syntax and semantics of datastore content, operational data and protocol operations. REST operations are used to access the hierarchical data within a datastore. Request and response data can be in XML or JSON format, where XML is the default format.

Note XML OR JSON FORMAT? Both XML and JSON are supported formats for requests and response data, but using JSON for configuration is currently not as mature and clearly defined as XML - which is used by NETCONF. It is therefore recommend to use XML encoding

To get a quick introduction to how to enable REST in NSO and how to run the CRUD operations, continue to the section called “Getting started”.

To read more about resources, continue to the section called “Resources”.

Tip All REST request/response examples in the section called “Getting started” are based on the *$NCS_DIR/ examples.ncs/service-provider/mpls-vpn* example. Go to this example and follow the information in the `README` and `README.REST` files. All other sections request/response examples in this chapter are based on the *$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/13-rest* example.

Tip Sending REST requests using a browser is an easy way to test things. But using REST with Basic Authentication in the browser should not be used because of security considerations. Use the JSON-RPC login mechanism instead. Once authenticated, REST requests don't need Basic Authentication anymore. You can read more about the JSON login mechanism in the section called “Method login” in *NSO 4.4.2.3 Web UI*.

Getting started

In order to enable REST in NSO, REST must be enabled in `ncs.conf`. The web server configuration for REST is shared with the WebUI's config. However, the WebUI does not have to be enabled for REST to work.

Here's a minimal example of what is needed in the conf file:

Example 1. NSO configuration for REST

```
<rest>
  <enabled>true</enabled>
</rest>

<webui>
  <enabled>false</enabled>
  <transport>
    <tcp>
      <enabled>true</enabled>
      <ip>0.0.0.0</ip>
      <port>8080</port>
    </tcp>
  </transport>
</webui>
```

Request URI structure

Resources are represented with URIs following the structure for generic URIs in [RFC3986].

Example 2. Request URI structure

```
 <OP> /api/<path>?<query>#<fragment>

  ^    ^     ^        ^       ^
  |    |     |        |       |
method entry resource query fragment

  M      M      O       O     I

M=mandatory, O=optional, I=ignored
```

```
<text> replaced by client with real values
```

A REST operation is derived from the HTTP method and the request URI, using the following conceptual fields:

- "method": the HTTP method identifying the REST operation requested by the client, to act upon the target resource specified in the request URI.
- "entry": the well-known REST entry point ("/api").

Note *THIS DIFFERS FROM RESTCONF!*

- "resource": the path expression identifying the resource that is being accessed by the operation. If this field is not present, then the target resource is the API itself, represented by the media type "application/vnd.yang.api".

Note *THE MEDIA TYPE DIFFERS FROM RESTCONF!*

- "query": the set of parameters associated with the REST message. These have the familiar form of "name=value" pairs. There is a specific set of parameters defined, see the section called “Query Parameters”. The contents of the query parameter value must be encoded according to [RFC2396], section 3.4. Any reserved characters must be encoded with escape sequences, according to [RFC2396], section 2.4.
- "fragment": This field is not used by the REST protocol.

The REST protocol uses HTTP methods to identify the CRUD operation requested for a particular resource. The following table shows how the REST operations relate to NETCONF protocol operations:

Table 3. REST vs NETCONF operations

REST	NETCONF
GET	<get-config>, <get>
POST	<edit-config> (operation="create")
PUT	<edit-config> (operation="replace")
PATCH	<edit-config> (operation="merge")
DELETE	<edit-config> (operation="delete")
OPTIONS	none
HEAD	none

Accessing the REST API

The REST API can be accessed e.g. by using the command line tool **curl**:

Example 4. Using curl for accessing NSO

```
$ curl -u admin:admin http://localhost:8080/api -X GET
```

To provide an HTTP header use the `-H` switch:

```
... -H "Accept: application/vnd.yang.api+xml"
```

By default curl will display responses on standard output. The headers are not included. To include these the "-i" switch has to be added.

`curl -i ...`

The full command with response:

```
$ curl -i -u admin:admin http://localhost:8080/api -X GET
  -H "Accept: application/vnd.yang.api+xml"

HTTP/1.1 200 OK
Server: NCS/3.4
Date: Tue, 03 Feb 2015 09:53:23 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Content-Length: 181
Content-Type: application/vnd.yang.api+xml
Vary: Accept-Encoding
Pragma: no-cache

<api xmlns="http://tail-f.com/ns/rest"
xmlns:y="http://tail-f.com/ns/rest">
  <version>0.5</version>
  <config/>
  <running/>
  <operational/>
  <operations/>
  <rollbacks/>
</api>
```

GET

The GET method is sent by the client to retrieve data and meta-data for a resource. It is supported for all resource types, except operation resources. The request must contain a request URI that contains at least the entry point component (`/api`). Note the use of the `Accept` HTTP header to indicate what format the returned result will be in. The value of the `Accept` HTTP header, in this example: `application/vnd.yang.data+json`, must be a valid media type. XML is the default format, so it is needed to explicitly request JSON in the example below. To read more about the media types, see the section called “Resources”.

In the following example all HTTP headers with the response are included, and both the XML and JSON format is shown. In later examples in this chapter parts of the headers can be omitted for brevity, and the XML format is used.

Example 5. Get the VPN `volvo`, "api/running/vpn/l3vpn/volvo", resource represented as JSON

```
$ curl -i -u admin:admin http://localhost:8080/api/running/vpn/l3vpn/volvo
  -X GET -H "Accept: application/vnd.yang.data+json"
```

The server might respond:

```
HTTP/1.1 200 OK
Server: NCS/3.4
Date: Tue, 03 Feb 2015 10:48:23 GMT
Last-Modified: Tue, 03 Feb 2015 10:36:36 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Etag: 1422-959797-517732
Content-Type: application/vnd.yang.data+json
Transfer-Encoding: chunked
Pragma: no-cache
```

```
{
  "l3vpn:l3vpn": {
    "name": "volvo",
    "as-number": 12345,
    "endpoint": [
      {
        "id": "branch-office1"
      },
      {
        "id": "branch-office2"
      },
      {
        "id": "main-office"
      }
    ]
  }
}
```

The returned l3vpn list instance `volvo` is prefixed with a module name - `l3vpn` - to indicate a particular YANG namespace.

To read more about the `ETag` and `Last-Modified` response headers, see the section called “Request/ Response headers”.

Example 6. Get the VPN `volvo`, "api/running/vpn/l3vpn/volvo", resource represented as XML

Note The query parameter `deep` is added to get the full subtree under the resource. See: the section called “Query Parameters” [34]

```
$ curl -i -u admin:admin http://localhost:8080/api/running/vpn/l3vpn/volvo?deep
  -X GET -H "Accept: application/vnd.yang.data+xml"
```

The server might respond:

```
HTTP/1.1 200 OK
...
<l3vpn xmlns="http://com/example/l3vpn"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:l3vpn="http://com/example/l3vpn">
  <name>volvo</name>
  <as-number>12345</as-number>
  <endpoint>
    <id>branch-office1</id>
    <ce-device>ce1</ce-device>
    <ce-interface>GigabitEthernet0/11</ce-interface>
    <ip-network>10.7.7.0/24</ip-network>
    <bandwidth>6000000</bandwidth>
  </endpoint>
  <endpoint>
    <id>branch-office2</id>
    <ce-device>ce4</ce-device>
    <ce-interface>GigabitEthernet0/18</ce-interface>
    <ip-network>10.8.8.0/24</ip-network>
    <bandwidth>6000000</bandwidth>
  </endpoint>
  <endpoint>
    <id>main-office</id>
    <ce-device>ce0</ce-device>
    <ce-interface>GigabitEthernet0/11</ce-interface>
    <ip-network>10.10.1.0/24</ip-network>
    <bandwidth>6000000</bandwidth>
  </endpoint>
</l3vpn>
```

Refer to the section called "GET and Query examples" for more resource retrieval examples.

POST

The POST method is sent by the client to create a data resource or invoke an operation resource ("rpc" or "tailf:action").

This is an example of resource creation using POST, followed by an example using POST to invoke an operation.

Example 7. Create the new VPN `volvo`, "api/running/vpn/l3vpn/volvo", resource, with xml payload

```
$ curl -i -u admin:admin http://localhost:8080/api/running/vpn -X POST
  -H "Content-Type: application/vnd.yang.data+xml"
  -T volvo-vpn.xml
```

If the resource is created, the server might respond as follows:

```
HTTP/1.1 201 Created
Server: NCS/3.4
Location: http://localhost:8080/api/running/vpn/l3vpn/volvo
...
```

The file `volvo-vpn.xml` contains the configuration for the VPN instance, for example similar to the configuration output from Example 6, "Get the VPN `volvo`, "api/running/vpn/l3vpn/volvo", resource represented as XML"

If the POST method succeeds, a `"201 Created"` Status-Line is returned and there is no response message body. Also, a `"Location"` header identifying the child resource that was created is present in the response.

Refer to the section called "Create a List Instance with POST" for more examples of creating resources.

If the target resource type is an operation resource, then the POST method is treated as a request to invoke that operation. The message body (if any) is processed as the operation input parameters.

Example 8. Invoke the operation "check-sync" for the VPN `volvo`, "api/running/vpn/l3vpn/volvo", resource

```
$ curl -i -u admin:admin
  http://localhost:8080/api/running/vpn/l3vpn/volvo/_operations/check-sync
  -X POST
  -H "Content-Type: application/vnd.yang.data+xml"
```

The server might respond as follows:

```
HTTP/1.1 200 OK
Server: NCS/3.4
Date: Tue, 03 Feb 2015 14:26:16 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Content-Length: 78
```

```
Content-Type: application/vnd.yang.operation+xml
Vary: Accept-Encoding
Pragma: no-cache

<output xmlns='http://com/example/l3vpn'>
  <in-sync>true</in-sync>
</output>
```

Refer to the section called “Operations and Actions” and the section called “Invoke Operations” for details on operation resources.

PUT

The PUT method is sent by the client to create or replace the target resource. The request must contain a request URI that contains a target resource that identifies the data resource to create or replace.

Example 9. Replace the VPN `volvo`, "api/running/vpn/l3vpn/volvo", resource contents

```
$ curl -i -u admin:admin http://localhost:8080/api/running/vpn/l3vpn/volvo -X PUT
  -H "Content-Type: application/vnd.yang.data+xml"
  -T volvo-vpn.xml
```

If the resource is updated, the server might respond:

```
HTTP/1.1 204 No Content
...
```

The file `volvo-vpn.xml` contains the, possible updated, configuration for the VPN instance.

If the PUT method creates a new resource, a "201 Created" Status-Line is returned. If an existing resource is modified, either "200 OK" or "204 No Content" is returned.

The "insert" and "resource" query parameters are supported by the PUT method for data resources, for more examples see the section called “Insert Data into Resources”.

Note The "resource" query parameter correspond to the "point" query parameter in RESTCONF. *THIS DIFFERS FROM RESTCONF!*

Refer to the section called “Create and Replace a List Instance with PUT” for more examples on how to use PUT.

PATCH

The PATCH method is used to create or update a sub-resource within the target resource. If the target resource instance does not exist, the server *WILL NOT* create it.

To replace just the `as-number` field in the `volvo` VPN (instead of replacing the entire resource with the PUT method), the client might send a plain patch as follows.

Example 10. Update the VPN `volvo`, "api/running/vpn/l3vpn/volvo/", resource contents

```
$ curl -i -u admin:admin http://localhost:8080/api/running/vpn/l3vpn/volvo/as-number
  -X PATCH -H "Content-Type: application/vnd.yang.data+xml"
  --data "<as-number>54321</as-number>"
```

If the resource is updated, the server might respond:

```
HTTP/1.1 204 No Content
...
```

Refer to the section called “Update Existing List Instance with PATCH” for more examples on how to use PATCH.

DELETE

The DELETE method is used to delete the target resource. If the DELETE method succeeds, a "204 No Content" Status-Line is returned, and there is no response message body.

Example 11. Delete the VPN `volvo`, "api/running/vpn/l3vpn/volvo", resource contents

```
$ curl -i -u admin:admin http://localhost:8080/api/running/vpn/l3vpn/volvo
  -X DELETE -H "Accept: application/vnd.yang.data+xml"
```

If the resource is successfully deleted, the server will respond:

```
HTTP/1.1 204 No Content
...
```

Refer to the section called “Delete Data Resources” for more examples on how to use DELETE.

OPTIONS

The OPTIONS method is sent by the client to discover which methods are supported by the server for a specific resource. The supported methods are listed in the ALLOW header.

Example 12. Get options for the VPN `volvo`, "api/running/vpn/l3vpn/volvo", resource

```
$ curl -i -u admin:admin http://localhost:8080/api/running/vpn/l3vpn/volvo
  -X OPTIONS -H "Accept: application/vnd.yang.data+xml"
```

The resource might respond:

```
HTTP/1.1 200 OK
Server: NCS/3.4
Allow: DELETE, GET, HEAD, PATCH, POST, PUT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Content-Length: 0
Content-Type: text/html
Pragma: no-cache
```

Here the options method responds with an ALLOW header indicating that all methods are allowed on the "volvo" resource.

HEAD

The HEAD method is sent by the client to retrieve just the headers that would be returned for the comparable GET method, without the response body. The access control behavior is enforced as if the method was GET instead of HEAD. The server will respond the same as if the method was GET instead of HEAD, except that no response body is included.

Example 13. HEAD for the VPN `volvo`, "api/running/vpn/l3vpn/volvo", resource

```
$ curl -i -u admin:admin http://localhost:8080/api/running/vpn/l3vpn/volvo
  -X HEAD -H "Accept: application/vnd.yang.data+xml"
```

The resource might respond:

```
HTTP/1.1 200 OK
Server: NCS/3.4
Date: Tue, 03 Feb 2015 13:43:23 GMT
Last-Modified: Tue, 03 Feb 2015 13:32:17 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Etag: 1422-970337-656224
Content-Length: 0
Content-Type: application/vnd.yang.data+xml
Pragma: no-cache
```

Here the "Content-Length" header is 0. This is because the content length calculation is not eligible when the actual content is suppressed as with the HEAD method.

Query Parameters

Each REST operation allows zero or more query parameters to be present in the request URI. The specific parameters that are allowed depends on the resource type, and sometimes the specific target resource used, in the request.

Table 14. Query Parameters

Name	Methods	Description
async-commit-queue	POST	If some device is non-operational or has data waiting in the commit queue, the data in this transaction will be placed in the commit queue.
deep	GET	Retrieve a resource with all subresources inline.
dryrun	POST	No data is sent to the devices. Instead the effects that would have taken place is showed in the returned diff output. Possible values are: *xml*, *cli* and *native*. The value used specify in what format we want the returned diff to be.
insert	POST	Used to specify where a resource should be inserted in a ordered-by user list. This query parameter is used together with the *resource* query parameter. Possible values are: *after*, *before*,*first* and *last*. See the section called “Insert Data into Resources” for details.
limit	GET	Used by the client to specify a limited set of list entries to retrieve. See the section called “Partial Responses” for details.
no-networking	POST	Do not send any data to the devices. This is a way to manipulate CDB in NSO without generating any southbound traffic.
no-out-of-sync-check	POST	Continue with the transaction even if NSO detects that a device's configuration is out of sync.
no-overwrite	POST	NSO will check that the data that should be modified has not changed on the device compared to NSO's view of the data.

Name	Methods	Description
offset	GET	Used by the client to specify a limited set of list entries to retrieve. See the section called “Partial Responses” for details.
operations	GET	Used by the client to include/exclude operations (tailf:actions) in the result. Possible values are: *true*, which is the default value, and *false*.
resource	POST	Used to specify where a resource should be inserted in a ordered-by user list. This query parameter is used together with the *insert* query parameter. See the section called “Insert Data into Resources” for details.
rollback-comment	POST	Used to specify a comment to be attached to the Rollback File that will be created as a result of the POST operation. This assume that Rollback File handling is enabled.
rollback-label	POST	Used to specify a label to be attached to the Rollback File that will be created as a result of the POST operation. This assume that Rollback File handling is enabled.
select	GET	Used by the client to select which nodes and subresources in a resource to retrieve. See the section called “Partial Responses” for details.
shallow	GET	Retrieve a resource with no subresources inline.
sync-commit-queue	POST	If some device is non-operational or has data waiting in the commit queue, the data in this transaction will be placed in the commit queue. The flag will cause the operation to not return until the transaction has been sent to all devices, or a timeout occurs.
unhide	GET	Used by the client to unhide hidden nodes. See the section called “Hidden Nodes” for details.
verbose	GET	Used by the client to control display of the "self" and "path" attributes of the resource. See the section called “Displaying Default Data” for details.
with-defaults	GET	Used by the client to control display of default data in GET requests. See the section called “Displaying Default Data” for details.

Note The query parameters of the REST API is not the same as for RESTCONF. *THIS DIFFERS FROM RESTCONF!*

Note The commit flags "no-overwrite" and "no-out-of-sync-check" will override their corresponding global settings, see also: "NSO Getting Started Guide", section: "RunTime Configuration". For REST examples, see: the section called “Making use of the commit flags”.

If neither "deep" nor "shallow" is used, a variable depth is returned. The output shown will stop at the first encountered presence container or list key value(s).

Resource Examples

Note In the following examples the curl calls are shortened to:

```
GET /<datastore>/<path>
Accept: application/vnd.yang...
```

Tip All REST request/response examples in this section is based on the *$NCS_DIR/examples.ncs/getting-started/developing-with-ncs/13-rest* example. Go to this example and follow the information in the README_router file to play around and get familiar with the REST API.

GET and Query examples

Retrieve Data Resources

If a resource only needs to be outlined, the *shallow* query parameter can be used on the GET request.

Example 15. Shallow get for the "sys" resource

```
GET /running/devices/device/ex0/config/sys?shallow
```

The server might respond:

```
<sys xmlns="http://example.com/router"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:r="http://example.com/router"
xmlns:ncs="http://tail-f.com/ns/ncs">
  <interfaces/>
  <routes/>
  <syslog/>
  <ntp/>
  <dns/>
</sys>
```

On the other hand, if the full subtree under a resource is required, the *deep* query parameter can be used on the GET request.

Example 16. Deep get for the "sys/interfaces/interface" resource

```
GET /running/devices/device/ex0/config/sys/interfaces/interface?deep
```

The server might respond:

```
<collection xmlns:y="http://tail-f.com/ns/rest">
  <interface xmlns="http://example.com/router">
    <name>eth0</name>
    <unit xmlns="http://example.com/router">
      <name>0</name>
      <enabled>true</enabled>
      <status xmlns="http://example.com/router">
```

```
        </status>
        <family xmlns="http://example.com/router">
          <inet xmlns="http://example.com/router">
            <address xmlns="http://example.com/router">
              <name>192.168.1.2</name>
              <prefix-length>16</prefix-length>
            </address>
          </inet>
        </family>
      </unit>
      <unit xmlns="http://example.com/router">
        <name>1</name>
        <enabled>true</enabled>
        <status xmlns="http://example.com/router">
        </status>
        <family xmlns="http://example.com/router">
          <inet xmlns="http://example.com/router">
            <address xmlns="http://example.com/router">
              <name>192.168.1.3</name>
              <prefix-length>16</prefix-length>
            </address>
          </inet>
        </family>
      </unit>
      <unit xmlns="http://example.com/router">
        <name>2</name>
        <enabled>true</enabled>
        <description>My Vlan</description>
        <vlan-id>18</vlan-id>
        <status xmlns="http://example.com/router">
        </status>
      </unit>
    </interface>
</collection>
```

For more info about "deep" vs "shallow", see: the section called “Query Parameters” [34]

Partial Responses

By default, the server sends back the full representation of a resource after processing a request. For better performance, the server can be instructed to send only the nodes the client really needs in a partial response.

To request a partial response for a set of list entries, use the "offset" and "limit" query parameters to specify a limited set of entries to be returned.

For example, to retrieve only 2 entries from the `sys/routes/inet/route` list, issue the command:

Example 17. Limit the response

```
GET
/running/devices/device/ex0/config/sys/routes/inet/route?offset=3&limi
t=2
```

The following request retrieves 2 entries starting from entry 3 (the first entry is 0):

```
<collection xmlns:y="http://tail-f.com/ns/rest">
  <route xmlns="http://example.com/router">
    <name>10.40.0.0</name>
    <prefix-length>16</prefix-length>
    <description>Route 4</description>
    <next-hop xmlns="http://example.com/router">
      <name>192.168.10.4</name>
    </next-hop>
  </route>
  <route xmlns="http://example.com/router">
    <name>10.50.0.0</name>
    <prefix-length>16</prefix-length>
    <description>Route 5</description>
    <next-hop xmlns="http://example.com/router">
      <name>192.168.10.5</name>
    </next-hop>
  </route>
</collection>
```

To request a filtered partial response, use the "select" query parameter to specify the nodes and subresources to be returned.

- Use a semicolon-separated list to select multiple nodes.
- Use "a/b" to select a node "b" that is nested within node "a"; use "a/b/c" to select a node "c" nested within "b".
- Specify node subselectors to request only specific subnodes by placing expressions in parentheses "( )" after any selected node.
- To specify all subnodes of a specific node use the special wildcard notion within parentheses "(*)"

NOTE: "a/b/c;a/b/d" is equivalent to "a/b(c;d)"

The following request selects the routes name and next-hop/name nodes:

Example 18. Limit the response with select

```
GET
/running/devices/device/ex0/config/sys/routes/inet/route/10.20.0.0,16?
select=name;next-hop(name)
```

The server might respond:

```
<route xmlns="http://example.com/router"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:r="http://example.com/router"
xmlns:ncs="http://tail-f.com/ns/ncs">
  <name>10.20.0.0</name>
  <next-hop>
    <name>192.168.10.2</name>
  </next-hop>
</route>
```

Hidden Nodes

By default, hidden nodes are not visible in the REST interface. In order to unhide hidden nodes for retrieval or editing, clients can use the query parameter "unhide". The format of the "unhide" parameter is a comma-separated list of

```
<groupname>[;<password>]
```

As an example:

```
unhide=extra,debug;secret
```

This example unhides the normal group "extra" and the password-protected group "debug" with the password "secret".

Displaying Default Data

Normally, leaf nodes that are not set but has a default value are not displayed at a GET request. This behavior can be controlled by the use of the "with-defaults" query parameter. This parameter can take one of three values:

- "report-all": all data nodes are be reported, including any data nodes considered to be default data by the server.
- "explicit": a data node that has been explicitly set is reported. This is also the case when a data node is explicitly set to a value that coincide with the default value for the node.
- "report-all-tagged": In this mode the server returns all data nodes, just like the "report-all" mode, except a data node that is considered by the server to contain default data will include an attribute to indicate this condition.

A request with the "with-defaults" query parameter without a specified value will be interpreted as "report-all".

On a normal GET request of the "sys/ntp/server" resource, the default values are not retrieved:

Example 19. The "sys/ntp/server" list (no defaults)

```
GET /running/devices/device/ex0/config/sys/ntp/server
```

```
<collection xmlns:y="http://tail-f.com/ns/rest">
  <server xmlns="http://example.com/router">
    <name>10.2.3.4</name>
    <key>2</key>
  </server>
</collection>
```

By setting the query parameter `with-defaults=report-all` the default values will also be presented in the response.

Example 20. The "sys/ntp/server" list with all defaults

```
GET
/running/devices/device/ex0/config/sys/ntp/server?with-defaults=report
-all
```

```
<collection xmlns:y="http://tail-f.com/ns/rest">
  <server xmlns="http://example.com/router">
    <name>10.2.3.4</name>
    <enabled>true</enabled>
    <peer>false</peer>
    <version>4</version>
    <key>2</key>
  </server>
</collection>
```

Examples using POST, PUT and PATCH

Create a List Instance with POST

POST can be used to create a child resource to the resource specified by the uri. In this example a "route" list entry is created. The URI points to the parent container of the list. The payload in the POST request contain a "route" node that, at least, contain the list keys.

Note When creating a list entry all keys must be given in the payload.

Example 21. Creating a "sys/routes/inet/route" resource

```
POST /running/devices/device/ex0/config/sys/routes/inet
Content-Type: application/vnd.yang.data+xml


<route>
  <name>10.20.3.0</name>
  <prefix-length>24</prefix-length>
</route>
```

The server might respond:

```
HTTP/1.1 201
```

When an element is successfully created the HTTP status response is 201. If the element already existed the status response is 409.

Create a Presence Container, within a list, with POST

In this example "multilink" presence container is created. The URI points to the list instance that contains the presence container, i.e the URI ends with the key(s). The payload in the POST request contain a "multilink" node that contain a value for the leaf node "group".

Note It is not possible to create a non-presence container with POST, as non-presence containers per definition always exists.

Example 22. Creating a "sys/interfaces/serial/ppp0/multilink" resource

```
POST /running/devices/device/ex0/config/sys/interfaces/serial/ppp0
Content-Type: application/vnd.yang.data+xml | egrep
'(HTTP.*Created.*|Locat*)'


<multilink>
  <group>1</group>
</multilink>
```

The server might respond:

```
HTTP/1.1 201 Created
Location:
http://127.0.0.1:8080/api/running/devices/device/ex0/config/r:sys/inte
rfaces/ex:serial/ppp0/multilink
```

Create and Replace a List Instance with PUT

PUT can be used to create or replace a resource. No distinction is made in the prerequisites for PUT. If no resource existed it is created, but if it existed it is replaced. In the example, note how the URI ends with the keys.

Example 23. Creating a "route" resource using PUT

```
PUT
/running/devices/device/ex0/config/sys/routes/inet/route/10.30.3.0,24
Content-Type: application/vnd.yang.data+xml


<route>
  <name>10.30.3.0</name>
  <prefix-length>24</prefix-length>
  <next-hop>
    <name>192.168.4.4</name>
    <metric>100</metric>
  </next-hop>
</route>
```

The server might respond:

```
HTTP/1.1 201
```

When an element is successfully created the HTTP status response is 201. Make a GET to verify the "route" list after the PUT of the element.

Example 24. The "route" resource after creation

```
GET
/running/devices/device/ex0/config/sys/routes/inet/route/10.30.3.0,24


<route xmlns="http://example.com/router"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:r="http://example.com/router"
xmlns:ncs="http://tail-f.com/ns/ncs">
  <name>10.30.3.0</name>
  <prefix-length>24</prefix-length>
  <next-hop>
    <name>192.168.4.4</name>
  </next-hop>
</route>
```

Note The "metric" node was not in the above result from the GET! This has to do with the (non) use of "deep" and "shallow", see: the section called “Query Parameters” [34]

A replace of the same "route" element:

Example 25. Replacing a "route" resource using PUT

```
PUT
/running/devices/device/ex0/config/sys/routes/inet/route/10.30.3.0,24
Content-Type: application/vnd.yang.data+xml


<route>
```

```
  <name>10.30.3.0</name>
  <prefix-length>24</prefix-length>
  <next-hop>
    <name>192.168.3.1</name>
    <metric>100</metric>
  </next-hop>
  <next-hop>
    <name>192.168.4.2</name>
    <metric>200</metric>
  </next-hop>
</route>
```

The server might respond:

```
HTTP/1.1 204
```

The resource was replaced, not created, hence the 204 response. To verify that the element is replaced:

Example 26. The "route" resource after replace

```
GET
/running/devices/device/ex0/config/sys/routes/inet/route/10.30.3.0,24


<route xmlns="http://example.com/router"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:r="http://example.com/router"
xmlns:ncs="http://tail-f.com/ns/ncs">
  <name>10.30.3.0</name>
  <prefix-length>24</prefix-length>
  <next-hop>
    <name>192.168.3.1</name>
  </next-hop>
  <next-hop>
    <name>192.168.4.2</name>
  </next-hop>
</route>
```

Create and Replace Presence Container with PUT

In this example a "multilink" presence container is created. The uri points to the not yet existing URI for the presence container "multilink". The payload in the PUT request contain a "multilink" node that contain a value for the leaf node 'group'.

Example 27. Creating a "sys/interfaces/serial/ppp0/multilink" resource

```
PUT
/running/devices/device/ex0/config/sys/interfaces/serial/ppp0/multilin
k
Content-Type: application/vnd.yang.data+xml


<multilink>
  <group>1</group>
</multilink>
```

The server might respond:

```
HTTP/1.1 201
```

Any subsequent request to the same URI will replace the content of "multilink" with the new payload. But the response code will instead be 204, since the resource already exists.

Note Have in mind that PUT will replace everything with the provided payload. So in the example above, if the container "multilink" contained additional subnodes, other than "group", they would have been deleted as they were not present in the payload.

Replace Non-Presence Container with PUT

In this example a non-presence container is populated with subnode data. The URI points to the existing URI for the non-presence container "authentication", since a non-presence container always exist if its parent exist.

Example 28. Creating a "sys/interfaces/serial/ppp0/authentication" resource

```
PUT
/running/devices/device/ex0/config/sys/interfaces/serial/ppp0/authenti
cation
Content-Type: application/vnd.yang.data+xml


<authentication>
  <method>pap</method>
  <list-name>foobar</list-name>
</authentication>
```

The server might respond:

```
HTTP/1.1 204
```

Note Have in mind that PUT will replace everything with the provided payload. So in the example above, if the container "authentication" contained additional subnodes, they would have been deleted as they were not present in the payload.

Example 29. The "authentication" resource after replace

```
GET
/running/devices/device/ex0/config/sys/interfaces/serial/ppp0/authenti
cation


<authentication xmlns="http://example.com/example-serial"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:ex="http://example.com/example-serial"
xmlns:r="http://example.com/router"
xmlns:ncs="http://tail-f.com/ns/ncs">
  <method>pap</method>
  <list-name>foobar</list-name>
</authentication>
```

Update Existing List Instance with PATCH

To update an existing resource the PATCH method can be used. PATCH is only allowed on existing resources. When using PATCH the payload should only contain the data that should be modified.

Example 30. Updating a "route" resource using PATCH

```
PATCH
/running/devices/device/ex0/config/sys/routes/inet/route/10.30.3.0,24
Content-Type: application/vnd.yang.data+xml


<route>
  <next-hop>
    <name>192.168.5.1</name>
    <metric>400</metric>
  </next-hop>
</route>
```

The server might respond:

```
HTTP/1.1 204
```

The response status 204 indicated successful update. To verify that a new next-hop entry has been added to the route:

Example 31. The "route" resource after update

```
GET
/running/devices/device/ex0/config/sys/routes/inet/route/10.30.3.0,24


<route xmlns="http://example.com/router"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:r="http://example.com/router"
xmlns:ncs="http://tail-f.com/ns/ncs">
  <name>10.30.3.0</name>
  <prefix-length>24</prefix-length>
  <next-hop>
    <name>192.168.3.1</name>
  </next-hop>
  <next-hop>
    <name>192.168.4.2</name>
  </next-hop>
  <next-hop>
    <name>192.168.5.1</name>
  </next-hop>
</route>
```

Update Presence Container with PATCH

PATCH can not be used to create new resources directly, since it must operate on existing resources. PATCH, in contrast to PUT, only merges the provided payload with the existing configuration, which makes it possible to create child resources within the target resource. In this example a "multilink" presence container is created. The uri points to the existing parent resource for the presence container "multilink".

Example 32. Creating a "sys/interfaces/serial/ppp0/multilink" resource

```
PATCH /running/devices/device/ex0/config/sys/interfaces/serial/ppp0
Content-Type: application/vnd.yang.data+xml


<serial>
  <multilink>
    <group>1</group>
  </multilink>
</serial>
```

The server might respond:

```
HTTP/1.1 204
```

Update Non-Presence Container with PATCH

In this example a non-presence container is populated with subnode data. The URI points to the existing URI for the non-presence container "authentication", since a non-presence container always exist if its parent exist.

Example 33. Creating a "sys/interfaces/serial/ppp0/authentication" resource

```
PATCH
/running/devices/device/ex0/config/sys/interfaces/serial/ppp0/authenti
cation
Content-Type: application/vnd.yang.data+xml


<authentication>
  <method>eap</method>
</authentication>
```

The server might respond:

```
HTTP/1.1 204
```

Note Have in mind that PATCH will merge the existing configuration with the provided payload. So in the example above, if the container "authentication" contained additional subnodes not present in the payload, they will remain in the resulting configuration. Below the node 'list-name' was not present in the payload, but is still present in the resulting configuration.

Example 34. The "authentication" resource after update

```
GET
/running/devices/device/ex0/config/sys/interfaces/serial/ppp0/authenti
cation

<authentication xmlns="http://example.com/example-serial"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:ex="http://example.com/example-serial"
xmlns:r="http://example.com/router"
xmlns:ncs="http://tail-f.com/ns/ncs">
  <method>eap</method>
  <list-name>foobar</list-name>
</authentication>
```

Insert Data into Resources

For an ordered-by user list, the POST request can include the query parameters *insert* and *resource*.

The *insert* parameter indicated where the new element should be created. Legal values are:

- *first*: insert on top of list.
- *last*: insert on bottom of list.
- *before*: insert before the element indicated by the *resource* parameter.
- *after*: insert after the element indicated by the *resource* parameter.

The *resource* parameter contains the uri to an existing element in the list.

Verify this functionality by first retrieving the "sys/dns/server" list:

Example 35. The "sys/dns/server" list before insert

```
GET /running/devices/device/ex0/config/sys/dns


<dns xmlns="http://example.com/router"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:r="http://example.com/router"
xmlns:ncs="http://tail-f.com/ns/ncs">
  <server>
    <address>10.2.3.4</address>
  </server>
</dns>
```

Insert a new element before the existing element:

Example 36. Insert=before in the "sys/dns/server" list

```
POST
/running/devices/device/ex0/config/sys/dns?insert=before&resource=/api
/running/devices/device/ex0/config/sys/dns/server/10.2.3.4
Content-Type: application/vnd.yang.data+xml


<server>
  <address>10.1.1.2</address>
</server>
```

The server might respond:

```
HTTP/1.1 201
```

When an element is successfully created the HTTP status response is 201 To verify the result by retreiving the list again:

Example 37. The "sys/dns/server" list after insert

```
GET /running/devices/device/ex0/config/sys/dns


<dns xmlns="http://example.com/router"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:r="http://example.com/router"
xmlns:ncs="http://tail-f.com/ns/ncs">
  <server>
    <address>10.1.1.2</address>
  </server>
  <server>
    <address>10.2.3.4</address>
  </server>
</dns>
```

Making use of the commit flags

The query parameters: "no-networking" , "no-overwrite" , "no-out-of-sync-check" and "async-commit-queue" doesn't take any values.

The "async-commit-queue" parameter will result in the client getting a commit queue `Id` in return.

The "sync-commit-queue" parameter may take an optional timeout value in form of a positive integer. This value represents the number of seconds to wait for the transaction to be committed. If the timeout expires, a commit queue `Id` in will be return.

The commit queue `Id` can be used for retrieving information about the commit queue item or to delete the item from the commit queue.

Example 38. Using the async-commit-queue flag

```
POST
/running/devices/device/ex0/config/sys/routes/inet?async-commit-queue
Content-Type: application/vnd.yang.data+json


HTTP/1.1 100 Continue
Server:
Allow: GET, POST, OPTIONS, HEAD
Content-Length: 0

HTTP/1.1 201 Created
Server:
Location:
http://127.0.0.1:8080/api/running/devices/device/ex0/config/r:sys/rout
es/inet/route/10.20.1.0,24
Date: Wed, 12 Aug 2015 11:51:00 GMT
Allow: GET, POST, OPTIONS, HEAD
Last-Modified: Wed, 12 Aug 2015 11:51:00 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Etag: 1439-380260-995039
Content-Length: 64
Content-Type: text/json
Vary: Accept-Encoding
Pragma: no-cache

{"commit-queue": {
  "id":25377281470,
  "status":"async"
  }
}
```

Invoke Operations

To invoke an operation, use the POST method. The message body (if any) is processed as the operation input parameters.

Example 39. An "archive-log" action request example

The following YANG model snippet shows the definition of the action *archive-log*:

```
  grouping syslog {
    list server {
      key "name";
      leaf name {
        type inet:host;
      }
      leaf enabled {
        type boolean;
      }
      list selector {
        key "name";
```

```
          leaf name {
            type int32;
          }
          leaf negate {
            type boolean;
          }
          leaf comparison {
            type enumeration {
              enum "same-or-higher";
              enum "same";
            }
          }
          leaf level {
            type syslogLevel;
          }
          leaf-list facility {
            type syslogFacility;
            min-elements 1;
            max-elements "8";
          }
        }
        leaf administrator {
          type string;
          tailf:hidden maint;
        }
        tailf:action archive-log {
          tailf:exec "./scripts/archive-log";
          input {
            leaf archive-path {
              type string;
            }
            leaf compress {
              type boolean;
            }
          }
          output {
            leaf result {
              type string;
            }
          }
        }
      }
    }
```

The action is invoked using the following uri. Note, the *_operations* tag that indicates the action invocation:

```
POST /running/devices/device/ex0/config/sys/syslog/server/10.3.4.5/
operations/archive-log
Content-Type: application/vnd.yang.data+xml


<input>
  <archive-path>/tmp</archive-path>
  <compress>false</compress>
</input>
```

The server might respond:

```
<output xmlns='http://example.com/router'>
  <result>success</result>
</output>
```

If the POST method succeeds, a "200 OK" Status-Line is returned if there is a response message body, and a "204 No Content" Status-Line is returned if there is no response message body.

If the user is not authorized to invoke the target operation, an error response containing a "403 Forbidden" Status-Line is returned to the client.

Delete Data Resources

A delete removes all data in the subtree under a resource. In this example the complete "sys/interfaces/ex:serial" list is deleted.

Example 40. delete the "sys/interfaces/ex:serial" list

```
DELETE /running/devices/device/ex0/config/sys/interfaces/ex:serial
```

The server might respond:

```
HTTP/1.1 204
```

The response status 204 indicates success. By retrieving the "sys/interfaces" resource it can verify that the "ex:serial" list is removed.

Example 41. The "sys/interfaces" resource after delete

```
GET /running/devices/device/ex0/config/sys/interfaces
```

```
<interfaces xmlns="http://example.com/router"
xmlns:y="http://tail-f.com/ns/rest"
xmlns:r="http://example.com/router"
xmlns:ncs="http://tail-f.com/ns/ncs">
  <interface>
    <name>eth0</name>
  </interface>
</interfaces>
```

Whenever a change is made and a rollback resource is created we can set a label and a comment for that change. If we used that together with the DELETE above, we could have used this command instead.

Example 42. delete the "sys/interfaces/ex:serial" list with rollback label and comment

```
DELETE /running/devices/device/ex0/config/sys/interfaces/ex:serial?rollback-c
omment=remove%20subtree&rollback-label=delete
```

Resources

The RESTCONF protocol operates on a hierarchy of resources, starting with the top-level API resource itself. Each resource represents a manageable component within the device.

A resource can be considered a collection of conceptual data and the set of allowed methods on that data. It can contain child nodes that are nested resources. The child resource types and methods allowed on them are data-model specific.

A resource has its own media type identifier, represented by the `Content-Type` header in the HTTP response message. A resource can contain zero or more nested resources. A resource can be created and deleted independently of its parent resource, as long as the parent resource exists.

The RESTCONF resources are accessed via a set of URIs defined in this document. The set of YANG modules supported by the server will determine the additional data model specific operations, top-level data node resources, and notification event messages supported by the server.

The resources used in the RESTCONF protocol are identified by the "path" component in the request URI, see Example 2, “Request URI structure”. Each operation is performed on a target resource.

Representation

The RESTCONF protocol defines some application specific media types to identify each of the available resource types. The following resource types are defined in the REST API:

Table 43. Resources and their Media Types

Resource	Media Type
API	application/vnd.yang.api
Datastore	application/vnd.yang.datastore
Data	application/vnd.yang.data
Collection	application/vnd.yang.collection
Operation	application/vnd.yang.operation

Note The REST API does not support all of the datastores defined in RESTCONF, neither does it use the same media types. *THIS DIFFERS FROM RESTCONF!*

XML representation

A resource is represented in XML as an XML element, with an XML attribute "y:self" that contains the URI for the resource. In the XML representation, every resource has an XML attribute:

```
y:self="..."
```

Leafs are properties of the resource. They are encoded in XML as elements.

XML namespaces must be used whenever there are multiple sibling nodes with the same local name. This only happens if a YANG module augments a node with the same name as another node in the same container or list. XML namespaces MAY always be used, even if there are no risk of a conflict.

JSON representation

In the JSON representation, this URI is encoded as:

```
"_self": "..."
```

In the representation of a list resource, the keys are always present, and encoded first.

JSON doesn't have anything similar to XML namespaces. However, the notion defined in the Internet Draft is adapted: *"JSON Encoding of Data Modeled with YANG"*, where a *<yang-module-name>:<tag>* tag name is used in the JSON data to indicate the namespace. Note that it is only when the namespace changes that this notion is used.

Example 44. Namespaces in JSON

```
...
{
  "tailf-ncs:device": {
      "name": "ce1",
      "address": "127.0.0.1",
      ...
      "config": {
        "tailf-ned-cisco-ios-xr:service": {
          ...
        },
        ...
      }
    }
  }
  ...
```

API Resource (/api)

The top-level resource has the media type "application/vnd.yang.api+xml" or "application/vnd.yang.api +json". It is accessible through the well-known URI "/api".

This resource has the following fields:

Table 45. Fields of the /api resource

Field	Description
version	The version of the REST api.
config	Link to the "config" resource.
running	Link to the "running" resource.
operational	Link to the "operational" resource.
operations	Container for available operations (i.e: YANG rpc statements).
rollbacks	Container for available rollback files.

In XML, this resource is represented as an XML document with the document root element "y:api", underneath which all the resource's fields are represented as subelements.

In JSON, this resource is represented as a JSON object.

Supported HTTP methods: GET, HEAD, OPTIONS.

The version Field

The "version" field is a string identifying the version of the REST API.

The config Resource

The "config" is just another name for the "running" datastore.

The media type of this resource is either "application/vnd.yang.datastore+xml" or "application/ vnd.yang.datastore+json".

Note This resource resembles the RESTCONF "Datastore Resource" except that it does not contain operational data.

The running Resource

The "running" resource represents the running configuration datastore, and is present on all devices. Not all devices support direct modification to this resource.

The media type of this resource is either "application/vnd.yang.datastore+xml" or "application/vnd.yang.datastore+json".

The operational Resource

The "operational" read-only resource represents the state data as well as the config data on the device, and is present on all devices. Note that actions defined as *config false* also will show up in this resource.

The media type of this resource is "application/vnd.yang.datastore+xml" or "application/vnd.yang.datastore+json".

Note This resource resembles the RESTCONF "Datastore Resource" except that it is read-only.

The transaction Resource

The "transaction" resource represents a transaction datastore created by the JSON-RPC API. This allows REST API requests that read or write towards an existing transaction, without committing the transaction (this is left up to other entities e.g. the JSON-RPC API).

The media type of this resource is either "application/vnd.yang.datastore+xml" or "application/vnd.yang.datastore+json".

The operations Container

The "operations" container contains all operations defined with the "rpc" statement in the YANG models supported on the device.

The rollbacks Container

The "rollbacks" container contains all available rollback files to be used to rollback the configuration state to an earlier incarnation.

The logout Resource

The "logout" read-only resource is a meta-resource that always replies with 401 Unauthorized in order to aid scenarios where the REST API credentials are being cached by the HTTP client (e.g. a browser). Calling this resource will prompt for new credentials on subsequent requests to any other resource on the same realm.

Examples

In order to retrieve the representation of this resource, a client can send:

Note the use of the 'verbose' query parameter, see the section called “Query Parameters”.

Example 46. GET the /api resource

```
GET /api?verbose
Application:vnd.yang.api+xml
```

The server might respond:

```
HTTP/1.1 200 OK
Server:
Date: Wed, 12 Aug 2015 11:51:42 GMT
```

```
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Content-Length: 288
Content-Type: application/vnd.yang.api+xml
Vary: Accept-Encoding
Pragma: no-cache

<api xmlns="http://tail-f.com/ns/rest"
xmlns:y="http://tail-f.com/ns/rest" y:self="/api">
  <version>0.5</version>
  <config y:self="/api/config"/>
  <running y:self="/api/running"/>
  <operational y:self="/api/operational"/>
  <operations/>
  <rollbacks y:self="/api/rollbacks"/>
</api>
```

Datastore Resource (/api/<datastore>)

The media types "application/vnd.yang.datastore+xml" and "application/vnd.yang.datastore+json" represent a complete datastore. All three configuration datastores defined by NETCONF are available, as the resources:

- running
- operational (read-only)

Note The state data defined by NETCONF is available as a read-only resource "operational". *THIS DIFFERS FROM RESTCONF!*

A *config* datastore (i.e the same as "running") resource has the following fields:

Table 47. Fields of the /api/<datastore> resource

Field	Description
operations	Container for available built-in operations.
y:operations	Container for available user defined actions.
<all top-level data models nodes>	Top-level nodes from the YANG models.

The *"operational"* datastore contains both *operational* and *config* data; as well as the containers as shown in the table: Table 47, "Fields of the /api/<datastore> resource".

In XML, the */api/<datastore>* resource is represented as an XML document with the document root element "y:data", underneath which all the resource's fields are represented as subelements.

In JSON, this resource is represented as a JSON object.

The operations are described below, and all represented by the media type "application/vnd.yang.operation", see the section called "Invoke Operations".

Examples

The examples in this section could instead have been performed using JSON specific mime types such as "application/vnd.yang.api+json", "application/vnd.yang.datastore+json" and "application/vnd.yang.data+json".

To retrieve a representation of the running datastore in XML format, a client can send:

(Note the use of the 'verbose' query parameter, see the section called “Query Parameters”)

Example 48. GET the /api/running resource

```
GET /api/running?verbose
Application:vnd.yang.api+xml
```

The server might respond:

```
HTTP/1.1 200 OK
Server:
Date: Wed, 12 Aug 2015 11:51:42 GMT
Last-Modified: Wed, 12 Aug 2015 11:51:41 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Etag: 1439-380301-692799
Content-Type: application/vnd.yang.datastore+xml
Transfer-Encoding: chunked
Pragma: no-cache

<data xmlns:y="http://tail-f.com/ns/rest" y:self="/api/running">
 ...
 ...
  <devices xmlns="http://tail-f.com/ns/ncs"
y:self="/api/running/devices">
 ...
    <device y:self="/api/running/devices/device/ex0">
      <name>ex0</name>
    </device>
    <device y:self="/api/running/devices/device/ex1">
      <name>ex1</name>
    </device>
    <device y:self="/api/running/devices/device/ex2">
      <name>ex2</name>
    </device>
    <y:operations y:path="/devices">
      <connect>/api/running/devices/_operations/connect</connect>
      <sync>/api/running/devices/_operations/sync</sync>
      <sync-to>/api/running/devices/_operations/sync-to</sync-to>

<sync-from>/api/running/devices/_operations/sync-from</sync-from>

<disconnect>/api/running/devices/_operations/disconnect</disconnect>

<check-sync>/api/running/devices/_operations/check-sync</check-sync>

<check-yang-modules>/api/running/devices/_operations/check-yang-module
s</check-yang-modules>

<fetch-ssh-host-keys>/api/running/devices/_operations/fetch-ssh-host-k
eys</fetch-ssh-host-keys>

<clear-trace>/api/running/devices/_operations/clear-trace</clear-trace
>
    </y:operations>
  </devices>
  <python-vm xmlns="http://tail-f.com/ns/ncs"
y:self="/api/running/python-vm">
    <logging y:self="/api/running/python-vm/logging">
      <log-file-prefix>./logs/ncs-python-vm</log-file-prefix>
    </logging>
    <y:operations y:path="/python-vm">
      <stop>/api/running/python-vm/_operations/stop</stop>
      <start>/api/running/python-vm/_operations/start</start>
    </y:operations>
  </python-vm>
  <services xmlns="http://tail-f.com/ns/ncs"
y:self="/api/running/services">
    <y:operations y:path="/services">

<check-sync>/api/running/services/_operations/check-sync</check-sync>

<commit-dry-run>/api/running/services/_operations/commit-dry-run</comm
it-dry-run>
    </y:operations>
  </services>
  <operations>
    <lock y:self="/api/running/_lock">/api/running/_lock</lock>
    <rollback
y:self="/api/running/_rollback">/api/running/_rollback</rollback>
  </operations>
</data>
```

Note that any action defined as "config false" will only show up under the */api/operational* datastore.

Data Resource (/api/<datastore>/<data>)

By default, all top-level objects, list entries, and containers are resources. Any resource derived from a YANG module is represented with the media type "application/vnd.yang.data+xml".

When such a resource is retrieved, a "path" property is included in its representation (a "y:path" attribute in XML). The value of this property is the resource's instance-identifier.

Supported HTTP methods: DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT.

Note that resources representing non-presence containers cannot be deleted, and thus they do not support the DELETE method.

Refer to the section called “Resource Examples” for examples of how to operate on "application/vnd.yang.data+xml".

Operations and Actions

YANG-defined operations, defined with the YANG statements "rpc" or "tailf:action", and the built-in operations, are represented with the media type "application/vnd.yang.operation".

Resources of this type accept only the method "POST".

In XML, such resources are encoded as subelements to the XML element "y:operations". In JSON, they are encoded under "_operations".

If an operation does not require any parameters, the POST message has no body. If the client wishes to send parameters to the operation, they are encoded as an XML document with the document element "input".

If an operation does not produce any output, the HTTP response code is 204 (No Content). If it produces output, the HTTP response code is 200 (OK), and the output of the operation is encoded as an XML document with the document element "output".

Supported HTTP methods: POST

The Rollback Resource

The rollback resource can be accessed from the top level "/api/rollbacks" resource as described above, and a rollback file can be applied to any database.

Listing and Inspecting Rollback Files

In order to list available a rollback files, a client can send:

Example 49. GET rollback files information

```
GET /api/rollbacks
Application:vnd.yang.api+xml
```

The server might respond:

```
<rollbacks xmlns="http://tail-f.com/ns/rest"
xmlns:y="http://tail-f.com/ns/rest">
  <file>
    <name>0</name>
    <creator>admin</creator>
    <date>2015-08-12 13:51:41</date>
    <via>rest</via>
    <label></label>
    <comment></comment>
  </file>
  <file>
    <name>1</name>
    <creator>admin</creator>
    <date>2015-08-12 13:51:41</date>
    <via>rest</via>
    <label></label>
    <comment></comment>
  </file>
</rollbacks>
```

Note how each rollback file is represented as separate resources, e.g. "/api/rollbacks/0". We can also see how the 'rollback-label' and 'rollback-comment' is used in "/api/rollbacks/0/label" and "/api/rollbacks/0/comment". These resources can be inspected individually and a client can send:

Example 50. GET rollback file content

```
GET /api/rollbacks/0
Application:vnd.yang.api+xml
```

The server might respond:

```
# Created by: admin
# Date: 2015-08-12 13:51:41
# Via: rest
# Type: delta
# Label:
# Comment:
# No: 10021

ncs:devices {
    ncs:device ex0 {
        ncs:config {
            r:sys {
                r:interfaces {
                    ex:serial ppp0 {
                        ex:ppp {
                            ex:accounting acme;
                        }
                        ex:authentication {
                            ex:method eap;
                            ex:list-name foobar;
                        }
                        ex:authorization admin;
                        ex:multilink {
                            ex:group 1;
                        }
                    }
                }
            }
        }
    }
}
```

The payload is in the same curly bracket rollback format as used in the NETCONF, CLI and Web UI agents.

Applying Rollback Files

To apply a rollback file to a database use the appropriate "rollback" resource/operation in the datastore:

Example 51. Find and use the rollback operation resource

```
GET /api/running
Application:vnd.yang.datastore+xml
```

The server might respond:

```
<data xmlns:y="http://tail-f.com/ns/rest">
 ...
 ...
  <devices xmlns="http://tail-f.com/ns/ncs">
 ...
    <device>
      <name>ex0</name>
    </device>
    <device>
      <name>ex1</name>
    </device>
    <device>
      <name>ex2</name>
    </device>
    <y:operations>
      <connect>/api/running/devices/_operations/connect</connect>
      <sync>/api/running/devices/_operations/sync</sync>
      <sync-to>/api/running/devices/_operations/sync-to</sync-to>

<sync-from>/api/running/devices/_operations/sync-from</sync-from>

<disconnect>/api/running/devices/_operations/disconnect</disconnect>

<check-sync>/api/running/devices/_operations/check-sync</check-sync>
```

```
<check-yang-modules>/api/running/devices/_operations/check-yang-module
s</check-yang-modules>

<fetch-ssh-host-keys>/api/running/devices/_operations/fetch-ssh-host-k
eys</fetch-ssh-host-keys>

<clear-trace>/api/running/devices/_operations/clear-trace</clear-trace
>
    </y:operations>
  </devices>
  <python-vm xmlns="http://tail-f.com/ns/ncs">
    <logging>
      <log-file-prefix>./logs/ncs-python-vm</log-file-prefix>
    </logging>
    <y:operations>
      <stop>/api/running/python-vm/_operations/stop</stop>
      <start>/api/running/python-vm/_operations/start</start>
    </y:operations>
  </python-vm>
  <services xmlns="http://tail-f.com/ns/ncs">
    <y:operations>

<check-sync>/api/running/services/_operations/check-sync</check-sync>

<commit-dry-run>/api/running/services/_operations/commit-dry-run</comm
it-dry-run>
    </y:operations>
  </services>
  <operations>
    <lock>/api/running/_lock</lock>
    <rollback>/api/running/_rollback</rollback>
  </operations>
</data>
```

Note the "/api/running/_rollback" resource operation.

POST an appropriate rollback file name to the "/api/running/_rollback" resource operation to apply it:

```
POST /api/running/_rollback
Content-Type: application/vnd.yang.data+xml
<file>0</file>
```

The server might respond:

```
HTTP/1.1 204
```

Configuration Meta-Data

It is possible to associate meta-data with the configuration data as attributes. For REST, resources such as containers, lists as well as leafs and leaf-lists can have such meta-data. For XML, this meta-data is represented as attributes, attached to the XML element in question. For JSON, there does not exist a natural way to represent this info. Hence a special notation is introduced, see the example below.

Example 52. XML representation of meta-data

```
<x xmlns="urn:x"
   y:self="/api/running/x"
   xmlns:y="http://tail-f.com/ns/rest"
   xmlns:x="urn:x"
   y:path="/x:x">
  <id tags=" important ethernet " annotation="hello world">42</id>
  <person y:self="/api/running/x/person" annotation="This is a person">
    <name>Bill</name>
    <person annotation="This is another person">grandma</person>
  </person>
</x>
```

Example 53. JSON representation of meta-data

```
{
  "x:x": {
    "_self": "/api/running/x",
    "_path": "/x:x",
    "id": 42,
    "@id": {"tags": ["important","ethernet"],"annotation": "hello world"},
    "person": {
      "_self": "/api/running/x/person",
      // NB: the below refers to the parent object
      "@@person": {"annotation": "This is a person"},
      "name": "Bill",
      "person": "grandma",
      // NB: the below refers to the sibling object
      "@person": {"annotation": "This is another person"}
    }
  }
}
```

For JSON, note how the meta data for a certain object "x" is represented by another object constructed of the object name prefixed with either one or two "@" signs. The meta-data object "@x" refers to the sibling object "x" and the "@@x" object refers to the parent object.

Note *THIS DIFFERS FROM RESTCONF!*

Request/Response headers

There are some optional request and response headers that are of interest since some functionality is obtained by their use. The focus here is on the `Etag` and `Last-Modified` response headers, and the request headers that are correlated to these (`If-Match`, `If-None-Match`, `If-Modified-Since` and `If-Unmodified-since`).

Response headers

- `Etag`: This header (entity-tag) is a string representing the latest transaction id in the database. This header is only available for the "running" resource or equivalent.
- `Last-Modified`: This header contains the timestamp for the last change in the database. This header is only available for the "running" resource or equivalent. Also, this header is only available if rollback files are enabled.

Request headers

- `If-None-Match`: This header evaluates to true if the supplied value does not match the latest `Etag` value. If evaluated to false an error response with status 304 (Not Modified) will be sent with no body. The usage of this is for instance for a GET operation to get information if the data has changed since last retrieval. This header carry only meaning if the `Etag` response header has previously been acquired.

- `If-Modified-Since`: This request-header field is used with a HTTP method to make it conditional, i.e if the requested variant has not been modified since the time specified in this field, an entity will not be returned from the server; instead, a 304 (Not Modified) response will be returned without any message-body.

  Usage of this is for instance for a GET operation to get information if (and only if) the data has changed since last retrieval. Thus, this header should use the value of a `Last-Modified` response header that has previously been acquired.
- `If-Match`: This header evaluates to true if the supplied value matches the latest `Etag` value. If evaluated to false an error response with status 412 (Precondition Failed) will be sent with no body.

  The usage of this can be to control if a POST operation should be executed or not (i.e. do not execute if the database has changed). This header carry only meaning if the `Etag` response header has previously been acquired.
- `If-Unmodified-Since`: This header evaluates to true if the supplied value is later or equal to the last acquired `Last-Modified` timestamp. If evaluated to false an error response with status 412 (Precondition Failed) will be sent with no body.

  The usage of this can be to control if a POST operation should be executed or not (i.e. do not execute if the database has changed). This header carry only meaning if the `Last-Modified` response header has previously been acquired.

Special characters

When setting or retrieving data it is sometimes necessary to represent special characters in the payload. In the REST API the payload can have both XML and JSON format. The special characters handled in the REST API are:

- "new line": representing a line feed (decimal ascii value 10)
- "carriage return": representing carriage return (decimal ascii value 13)
- "horizontal tab": representing a tabulation (decimal ascii value 9)

The ambition in the REST API is that special characters should be handled in the same way as they are in the CLI. Since the CLI is capable to present configuration data both as strings and XML the CLI representation can be used as template for both the XML and JSON format.

XML representation of special characters

When in the XML case the special characters uses the representation &#xH; where H is the ascii hex value or &#DD; where DD is the ascii decimal value. Since "&" is the quoting character this is also treated as a special character, and so is "<" since this is the separator character indicating the beginning of a tag value. The following is the list of XML special characters:

- "new line": represented by &#xA; or &#10;
- "carriage return": represented by &#xD; or &#13;
- "horizontal tab": represented by &#x9; or &#09;
- "&": represented by &amp;
- "<": represented by &lt;

An example XML fragment is

```
<name>123\n456\\n &lt;&amp;></name>

which is interpreted as:

123
456\n <&>
```

JSON representation of special characters

In the JSON formatted string case the quote character "\" and string separator characters are used "\"" which also becomes special characters in the string case. The complete list of JSON special characters become:

- "new line": represented by \n
- "carriage return": represented by \r
- "horizontal tab": represented by \t
- "\": represented by \ or \\
- """: represented by \"

The \ quote character is used in the following way: A single \ in a string that is not directly followed by characters n,r or t are unaltered (as a \ character). Two \\ are interpreted as \ (the escaped \). This implies that both \\\ and \\\\ are are interpreted as \\ and so forth.

An example JSON string is

```
"123\n456\\n \\\"

which is interpreted as:

123
456\n \\
```

Error Responses

Error responses are formatted either in XML and JSON depending on the preferred MIME type in the request.

In the installed NSO release there is a YANG file named `tailf-rest-error.yang` that defines the structure of these error replies. An easy way to find the file is to run, from the top directory of the NSO installation installation:

```
$ find . -name tailf-rest-error.yang
```

Example 54. Example of a XML formatted error message

```
$ curl -i -X PUT "localhost:8008/api/running/hosts"
  -H "Content-Type: application/vnd.yang.data+xml" ...

HTTP/1.1 500 Internal Server Error
Server: NCS/3.4
Cache-control: private, no-cache, must-revalidate, proxy-revalidate
Date: Thu, 10 Jul 2014 07:59:04 GMT
Content-Length: 259
Content-Type: text/xml
```

```
<errors xmlns="http://tail-f.com/ns/tailf-rest-error">
  <error>
    <error-tag>operation-failed</error-tag>
    <error-urlpath>/api/running/hosts</error-urlpath>
    <error-message>internal error</error-message>
  </error>
</errors>
```

Example 55. Example of a JSON formatted error message

```
$ curl -i -X POST "localhost:8008/api/running/hosts2"
  -H "Content-Type: application/vnd.yang.data+json" ...

HTTP/1.1 400 Bad Request
Server: NCS/3.4
Cache-control: private, no-cache, must-revalidate, proxy-revalidate
Date: Thu, 10 Jul 2014 09:11:13 GMT
Content-Length: 199
Content-Type: text/json

{
  "errors": {
    "error": [
      {
        "error-message": "unexpected trailing data: hosts",
        "error-urlpath": "/api/running/hosts",
        "error-tag": "malformed-message"
      }
    ]
  }
}
```

The YANG model for the error messages is taken from the NETCONF specification. However, note that the REST API currently only sets three values when reporting an error:

- "error-tag": An error classification tag.
- "error-urlpath": The URI used by the error generating request.
- "error-message": A descriptive error message.

The error-tag element has a number of predefined values and there is also a preferred HTTP status code connected to each error-tag. These are:

- "in-use": HTTP Status: `409 Conflict`
- "invalid-value": HTTP Status: `400 Bad Request`
- "too-big": HTTP Status: `413 Request Entity Too Large`
- "missing-attribute": HTTP Status: `400 Bad Request`
- "bad-attribute": HTTP Status: `400 Bad Request`
- "unknown-attribute": HTTP Status: `400 Bad Request`
- "bad-element": HTTP Status: `400 Bad Request`
- "unknown-element": HTTP Status: `400 Bad Request`
- "unknown-namespace": HTTP Status: `400 Bad Request`
- "access-denied": HTTP Status: `403 Forbidden`
- "lock-denied": HTTP Status: `409 Conflict`

- "resource-denied": HTTP Status: `409 Conflict`
- "rollback-failed": HTTP Status: `500 Internal Server Error`
- "data-exists": HTTP Status: `409 Conflict`
- "data-missing": HTTP Status: `409 Conflict`
- "operation-not-supported": HTTP Status: `501 Not Implemented`
- "operation-failed": HTTP Status: `500 Internal Server Error`
- "partial-operation": HTTP Status: `500 Internal Server Error`
- "malformed-message": HTTP Status: `400 Bad Request`
- "data-not-unique": HTTP Status: `400 Bad Request`

User extended error messages

For data providers, hooks, transforms etc there exist a possibility to add extensions to error messages and change error codes before sending errors back to the server from the callbacks. These codes and error messages will also be visible over the REST interface. More on how to use these options can be found in the confd_lib_dp(3) in *NSO 4.4.2.3 Manual Pages* man page, e.g under the `confd_db_seterr_extended` function, or in the Javadoc for the `com.tailf.dp.DpCallbackExtendedException` class.

Using the above mechanism to change the errorcode for an emitted error, will have effect on the REST HTTP response statuses. The following table show their relationship:

Table 56. Error code vs HTTP Status

Error Code	HTTP Status
CONFD_ERRCODE_IN_USE	409 Conflict
CONFD_ERRCODE_RESOURCE_DENIED	409 Conflict
CONFD_ERRCODE_INCONSISTENT_VALUE	400 Bad Request
CONFD_ERRCODE_ACCESS_DENIED	403 Forbidden
CONFD_ERRCODE_APPLICATION	400 Bad Request
CONFD_ERRCODE_APPLICATION_INTERNAL	500 Internal Server Error
CONFD_ERRCODE_DATA_MISSING	409 Conflict
CONFD_ERRCODE_INTERRUPT	500 Internal Server Error

The REST Query API

The Query API consists of a number of Requests and Replies which are sent as payload via the (REST) HTTP connection.

In the installed NSO release there are two Yang files named `tailf-rest-query.yang` and `tailf-common-query.yang` that defines the structure of these Requests / Replies. An easy way to find the file is to run, from the top directory of the NSO installation installation:

```
$ find . -name tailf-rest-query.yang
```

The API consists of the following Requests:

- `start-query`: Start a query and return a query handle.

- `fetch-query-result`: Use a query handle to repeatedly fetch chunks of the result.
- `reset-query`: (Re)set where the next fetched result will begin from.
- `stop-query`: Stop (and close) the query.

The API consists of the following Replies:

- `start-query-result`: Reply to the start-query request
- `query-result`: Reply to the fetch-query-result request

In the following examples, this data model is used:

```
container x {
  list host {
    key number;
    leaf number {
      type int32;
    }
    leaf enabled {
      type boolean;
    }
    leaf name {
      type string;
    }
    leaf address {
      type inet:ip-address;
    }
  }
}
```

The actual format of the payload should be represented either in XML or JSON. For XML it could look like this:

```
<start-query xmlns="http://tail-f.com/ns/tailf-rest-query">
  <foreach>
    /x/host[enabled = 'true']
  </foreach>
  <select>
    <label>Host name</label>
    <expression>name</expression>
    <result-type>string</result-type>
  </select>
  <select>
    <expression>address</expression>
    <result-type>string</result-type>
  </select>
  <sort-by>name</sort-by>
  <limit>100</limit>
  <offset>1</offset>
</start-query>
```

An informal interpretation of this query is:

For each '/x/host' where 'enabled' is true, select its 'name', and 'address', together with the specified 'label' and return the result sorted by 'name', in chunks of 100 results at the time.

When using XML, specify the name space as shown:

```
<start-query xmlns="http://tail-f.com/ns/tailf-rest-query">
```

The actual XPath query to run is specified by the 'foreach' element. In the example below will search for all '/x/host' nodes that has the 'enabled' node set to 'true':

```
<foreach>
  /x/host[enabled = 'true']
</foreach>
```

Define what to have returned from the node set by using one or more 'select' sections. What to actually return is defined by the XPath 'expression'.

Choose how the result should be represented. Basically, it can be the actual value or the path leading to the value. This is specified per select chunk The possible result-types are: `string` , `path` , `leaf-value` and `inline`.

The difference between `string` and `leaf-value` is somewhat subtle. In the case of `string` the result will be processed by the XPath function `string()` (which if the result is a node-set will concatenate all the values). The `leaf-value` will return the value of the first node in the result. As long as the result is a leaf node, `string` and `leaf-value` will return the same result. In the example above, we are using `string` as shown below. At least one `result-type` must be specified.

The result-type `inline` makes it possible to return the full sub-tree of data, either in XML or in JSON format. The data will be enclosed with a tag: `data`.

It is possible to specify an optional `label` for a convenient way of labeling the returned data:

```
<select>
  <label>Host name</label>
  <expression>name</expression>
  <result-type>string</result-type>
</select>
<select>
  <expression>address</expression>
  <result-type>string</result-type>
</select>
```

The returned result can be sorted. This is expressed as XPath expressions, which in most cases are very simple and refers to the found node set. In this example the result is sorted by the content of the 'name' node:

```
<sort-by>name</sort-by>
```

To limit the max amount of results in each chunk that `fetch-query-result` will return, set the 'limit' element. The default is to get all results in one chunk.

```
<limit>100</limit>
```

The 'offset' element specifies at which node to start receiving the result. The default is 1, i.e., the first node in the resulting node-set.

```
<offset>1</offset>
```

This request, expressed in JSON, would look like this:

```
{
 "start-query": {
   "foreach": "/x/host[enabled = 'true']",
   "select": [
     {
       "label": "Host name",
       "expression": "name",
       "result-type": ["string"]
     },
     {
       "expression": "address",
       "result-type": ["string"]
     }
   ],
   "sort-by": ["name"],
   "limit": 100,
   "offset": 1
 }
}
```

By putting the XML example in a file `test.xml` it can sent as a request, using the command 'curl', like this:

```
curl -i 'http://admin:admin@localhost:8008/api/query' \
     -X POST -T test.xml \
     -H "Content-Type: application/vnd.yang.data+xml"
```

The important parts of the above is the '/api/query' in the URI and the use of the HTTP 'POST' method with the correct 'Content-Type'.

The result would look something like this:

```
<start-query-result>
  <query-handle>12345</query-handle>
</start-query-result>
```

The query handle (in this example '12345') must be used in all subsequent calls. To retrieve the result, send:

```
<fetch-query-result xmlns="http://tail-f.com/ns/tailf-rest-query">
  <query-handle>12345</query-handle>
</fetch-query-result>
```

Which will result in something like the following:

```
<query-result xmlns="http://tail-f.com/ns/tailf-rest-query">
  <result>
    <select>
      <label>Host name</label>
      <value>One</value>
    </select>
    <select>
      <value>10.0.0.1</value>
    </select>
  </result>
  <result>
    <select>
      <label>Host name</label>
      <value>Three</value>
    </select>
    <select>
      <value>10.0.0.3</value>
    </select>
  </result>
</query-result>
```

Subsequent 'fetch-query-result' might get more 'result' entries in return until no more data exists and get an empty query result back:

```
<query-result xmlns="http://tail-f.com/ns/tailf-rest-query">
</query-result>
```

To go back in the "stream" of received data chunks and have them repeated, the `reset-query` request can be used. The example below asks to get results from the 42:nd result entry:

```
<reset-query xmlns=\"http://tail-f.com/ns/tailf-rest-query\">
  <query-handle>12345</query-handle>
  <offset>42</offset>
</reset-query>
```

To stop the query:

```
<stop-query xmlns="http://tail-f.com/ns/tailf-rest-query">
  <query-handle>12345</query-handle>
</stop-query>
```

Custom Response HTTP Headers

The REST server can be configured to reply with particular HTTP headers in the HTTP response. For example, to support Cross-Origin Resource Sharing (CORS, http://www.w3.org/TR/cors/) there is a need to add a couple of headers to the HTTP Response.

Add the extra configuration parameter in `ncs.conf`

Example 57. NSO configuration for REST

```
<rest>
  <enabled>true</enabled>
  <custom-headers>
    <header>
      <name>Access-Control-Allow-Origin</name>
      <value>*</value>
    </header>
  </custom-headers>
</rest>
```

Example 58.

Send a request with Origin header:

```
OPTIONS .bob.com
```

A result can then look like

```
HTTP/1.1 200 OK
Server:
Allow: GET, HEAD
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Content-Length: 0
Content-Type: text/html
Access-Control-Allow-Origin: http://api.bob.com
Pragma: no-cache
```

HTTP Status Codes

The REST server will return standard HTTP response codes, as described in the list below:

200 OK	The request was successfully completed, and a response body is returned containing a representation of the resource.
201 Created	A resource was created, and the new resource URI is returned in the "Location" header.

204 No Content	The request was successfully completed, but no response body is returned.
400 Bad Request	The request could not be processed because it contains missing or invalid information (such as validation error on an input field, a missing required value, and so on).
401 Unauthorized	The request requires user authentication. The response includes a "WWW-Authenticate" header field for basic authentication.
403 Forbidden	Access to the resource was denied by the server, due to authorization rules.
404 Not Found	The requested resource does not exist.
405 Method Not Allowed	The HTTP method specified in the request (DELETE, GET, HEAD, PATCH, POST, PUT) is not supported for this resource.
406 Not Acceptable	The resource identified by this request is not capable of generating the requested representation, specified in the "Accept" header or in the "format" query parameter.
409 Conflict	This code is used if a request tries to create a resource that already exists.
415 Unsupported Media Type	The format of the request is not supported.
500 Internal Error	The server encountered an unexpected condition which prevented it from fulfilling the request.
501 Not Implemented	The server does not (currently) support the functionality required to fulfill the request.
503 Unavailable	The server is currently unable to handle the request due to the resource being used by someone else, or the server is temporarily overloaded.

CHAPTER 4

The RESTCONF API

Introduction

RESTCONF is a HTTP based protocol, very similar to the NSO REST API. The NSO implementation of RESTCONF is based on the Internet-Draft named: *draft-ietf-netconf-restconf-17*.

This chapter describes any extensions and/or deviations between our implementation and the Internet-Draft.

Getting started

In order to enable RESTCONF in NSO, RESTCONF must be enabled in the `ncs.conf` configuration file. The web server configuration for RESTCONF is shared with the WebUI's config. However, the WebUI does not have to be enabled for RESTCONF to work.

Here's a minimal example of what is needed in the `ncs.conf` file:

Example 59. NSO configuration for REST

```
<restconf>
  <enabled>true</enabled>
</restconf>

<webui>
  <enabled>false</enabled>
  <transport>
    <tcp>
      <enabled>true</enabled>
      <ip>0.0.0.0</ip>
      <port>8080</port>
    </tcp>
  </transport>
</webui>
```

Root resource discovery

RESTCONF makes it possible to specify where the RESTCONF API is located, as described in the RESTCONF Internet-Draft.

As per default, the RESTCONF API root is */restconf*. To change this, configure the RESTCONF API root in the `ncs.conf` file as:

Example 60. NSO configuration for RESTCONF

```
<restconf>
  <enabled>true</enabled>
  <rootResource>my_own_restconf_root</rootResource>
</restconf>
```

The RESTCONF API root will now be */my_own_restconf_root*.

Note In this document, all examples will assume the RESTCONF API root to be */restconf*.

Extensions

To avoid any potential future conflict with the RESTCONF standard, any extensions made to the NSO implementation of RESTCONF is located under the URL path: */restconf/tailf*, or is controlled by means of a vendor specific media type.

Collections

The RESTCONF specification states that a result containing multiple instances (e.g a number of list entries) is not allowed if XML encoding is used. The reason for this is that an XML document can only have one root node.

To remedy this, a HTTP GET request can make use of the "Accept:" media type: *application/ vnd.yang.collection+xml* as shown in the following example.

Example 61. Use of collections

```
curl  -H "Accept: application/vnd.yang.collection+xml"  http://....
```

The result will then be wrapped with a "collection" element as shown below:

Example 62. Use of collections

```
<collection xmlns:y="urn:ietf:params:xml:ns:yang:ietf-restconf">
  ....
</collection>
```

Commit queue

The *commit queue* related query parameters that exist in the NSO REST API are also available in the NSO RESTCONF API, as described in the section called “Query Parameters”.

The query parameters are: *async-commit-queue*, *sync-commit-queue*, *no-networking*, *no-out-of-sync-check*, *no-overwrite*.

The RESTCONF Query API

Refer to the section called “The REST Query API” for a complete description of this functionality.

The only difference in the NSO implementation is the use of the RESTCONF URI root resource, and the RESTCONF media types. An example of how a **curl** request can look like is shown below:

```
curl -i 'http://admin:admin@localhost:8080/restconf/tailf/query' \
     -X POST -T test.xml \
     -H "Content-Type: application/yang-data+xml"
```

Deviations

The NSO RESTCONF server support all the mandatory parts of the specification and some optional parts. This chapter describes what is currently not supported and/or any other deviations from the specification.

Notifications

Currently, RESTCONF notifications are not supported.

Query Parameters

The following optional query parameters are currently not supported: *filter*, *start-time*, *stop-time*, *replay*.

CHAPTER 5

The NSO SNMP Agent

Introduction

The SNMP agent in NSO is used mainly for monitoring and notifications. It supports SNMPv1, SNMPv2c, and SNMPv3.

The following standard MIBs are supported by the SNMP agent:

- SNMPv2-MIB RFC 3418
- SNMP-FRAMEWORK-MIB RFC 3411
- SNMP-USER-BASED-SM-MIB RFC 3414
- SNMP-VIEW-BASED-ACM-MIB RFC 3415
- SNMP-COMMUNITY-MIB RFC 3584
- SNMP-TARGET-MIB and SNMP-NOTIFICATION-MIB RFC 3413
- SNMP-MPD-MIB RFC 3412
- TRANSPORT-ADDRESS-MIB RFC 3419
- SNMP-USM-AES-MIB RFC 3826
- IPV6-TC RFC 2465

Configuring the SNMP Agent

The SNMP agent is configured through any of the normal NSO northbound interfaces. It is possible to control most aspects of the agent through for example the CLI.

The YANG models describing all configuration capabilities of the SNMP agent reside under `$NCS_DIR/src/ncs/snmp/snmp-agent-config/*.yang` in the NSO distribution.

An example session configuring the SNMP agent through the CLI may look like:

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# snmp agent udp-port 3457
admin@ncs(config)# snmp community public name foobaz
admin@ncs(config-community-public)# commit
Commit complete.
```

```
admin@ncs(config-community-public)# top
admin@ncs(config)# show full-configuration snmp
snmp agent enabled
snmp agent ip    0.0.0.0
snmp agent udp-port 3457
snmp agent version v1
snmp agent version v2c
snmp agent version v3
snmp agent engine-id enterprise-number 32473
snmp agent engine-id from-text testing
snmp agent max-message-size 50000
snmp system contact ""
snmp system name ""
snmp system location ""
snmp usm local user initial
 auth sha password GoTellMom
 priv des password GoTellMom
!
snmp target monitor
 ip       127.0.0.1
 udp-port 162
 tag      [ monitor ]
 timeout  1500
 retries  3
 v2c sec-name public
!
snmp community public
 name     foobaz
 sec-name public
!
snmp notify foo
 tag  monitor
 type trap
!
snmp vacm group initial
 member initial
  sec-model [ usm ]
 !
 access usm no-auth-no-priv
  read-view   internet
  notify-view internet
 !
 access usm auth-no-priv
  read-view   internet
  notify-view internet
 !
 access usm auth-priv
  read-view   internet
  notify-view internet
 !
!
snmp vacm group public
 member public
  sec-model [ v1 v2c ]
 !
 access any no-auth-no-priv
  read-view   internet
  notify-view internet
 !
!
snmp vacm view internet
 subtree 1.3.6.1
```

```
  included
 !
!
snmp vacm view restricted
 subtree 1.3.6.1.6.3.11.2.1
  included
 !
 subtree 1.3.6.1.6.3.15.1.1
  included
 !
!
```

The SNMP agent configuration data is stored in CDB as any other configuration data, but is handled as a transformation between the data shown above and the data stored in the standard MIBs.

If you want to have a default configuration of the SNMP agent, you must provide that in an XML file. The initialization data of the SNMP agent is stored in an XML file that has precisely the same format as CDB initialization XML files, but it is not loaded by CDB, rather it is loaded at first startup by the SNMP agent. The XML file must be called `snmp_init.xml` and it must reside in the load path of NSO. In the NSO distribution, there is such an initialization file in `$NCS_DIR/etc/ncs/snmp/snmp_init.xml`. It is strongly recommended that this file is customized with another engine id and other community strings and v3 users.

If no `snmp_init.xml` file is found in the load path a default configuration with the agent disabled is loaded. Thus, the easiest way to start NSO without the SNMP agent is to ensure that the directory `$NCS_DIR/etc/ncs/snmp/` is not part of the NSO load path.

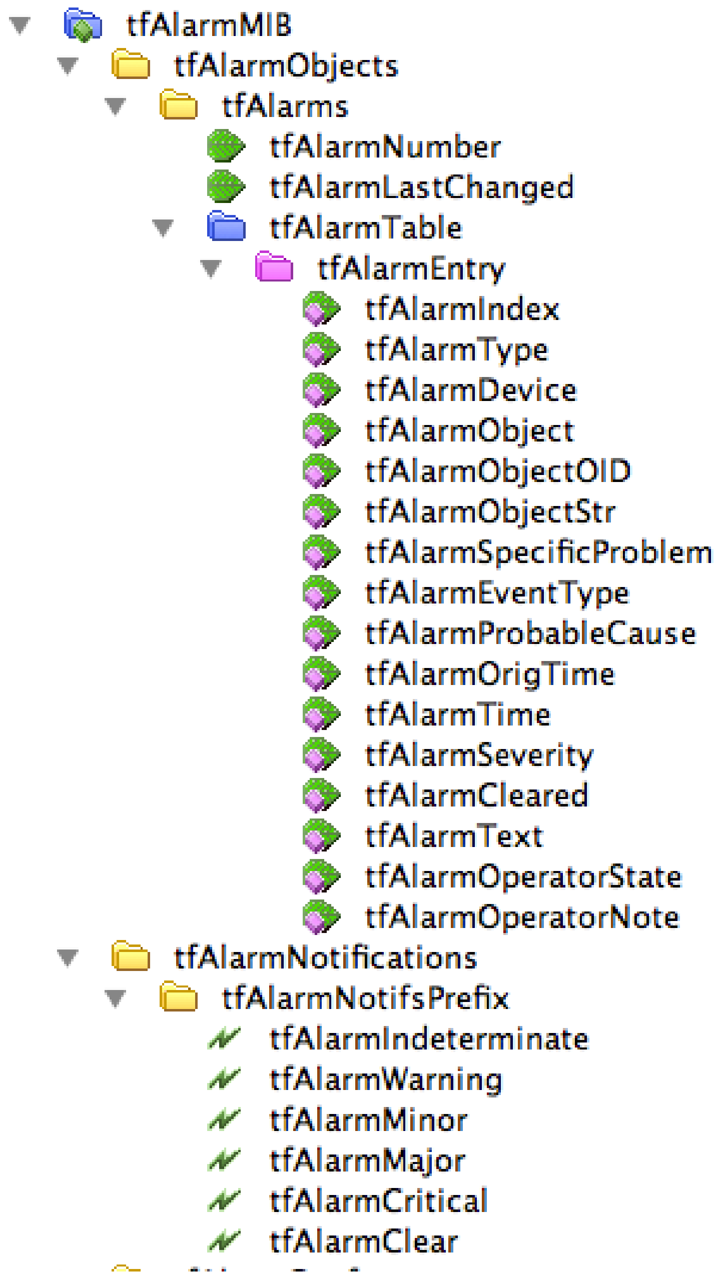
Note, this only relates to initialization the first time NSO is started. On subsequent starts, all the SNMP agent confiuration data is stored in CDB and the `snmp_init.xml` is never used again.

The Alarm MIB

The NSO SNMP alarm MIB is designed for ease of use in alarm systems. It defines a table of alarms and SNMP alarm notifications corresponding to alarm state changes. Based on the alarm model in NSO (see Chapter 6, *NSO Alarms*), the notifications as well as the alarm table contains the parameters that are required for alarm standards compliance (X.733 and 3GPP). The MIB files are located in `$NCS_DIR/src/ncs/snmp/mibs`.

TAILF-TOP-MIB.mib	the tail-f enterprise OID
TAILF-TC-MIB.mib	textual conventions for the alarm mib
TAILF-ALARM-MIB.mib	the actual alarm MIB
IANA-ITU-ALARM-TC-MIB.mib	import of IETF mapping of X.733 parameters
ITU-ALARM-TC-MIB.mib	import of IETF mapping of X.733 parameters

Figure 63. The NSO Alarm MIB



```
▼ tfAlarmMIB
    ▼ tfAlarmObjects
        ▼ tfAlarms
              tfAlarmNumber
              tfAlarmLastChanged
            ▼ tfAlarmTable
                ▼ tfAlarmEntry
                      tfAlarmIndex
                      tfAlarmType
                      tfAlarmDevice
                      tfAlarmObject
                      tfAlarmObjectOID
                      tfAlarmObjectStr
                      tfAlarmSpecificProblem
                      tfAlarmEventType
                      tfAlarmProbableCause
                      tfAlarmOrigTime
                      tfAlarmTime
                      tfAlarmSeverity
                      tfAlarmCleared
                      tfAlarmText
                      tfAlarmOperatorState
                      tfAlarmOperatorNote
    ▼ tfAlarmNotifications
        ▼ tfAlarmNotifsPrefix
              tfAlarmIndeterminate
              tfAlarmWarning
              tfAlarmMinor
              tfAlarmMajor
              tfAlarmCritical
              tfAlarmClear
```

The alarm table has the following columns:

tfAlarmIndex	an imaginary index for the alarm row that is persistent between restarts
tfAlarmType	this provides an identification of the alarm type and together with tfAlarmSpecificProblem forms a unique identification of the alarm
tfAlarmDevice	the alarming network device - can be NSO itself
tfAlarmObject	the alarming object within the device
tfAlarmObjectOID	in case the original alarm notification was a SNMP notification this column identifies the alarming SNMP object

tfAlarmObjectStr	name of alarm object based on any other naming
tfAlarmSpecificProblem	this object is used when the 'tfAlarmType' object cannot uniquely identify the alarm type
tfAlarmEventType	the event type according to X.733 and based on the mapping of the alarm type in the NSO alarm model
tfAlarmProbableCause	the probable cause to X.733 and based on the mapping of the alarm type in the NSO alarm model. Note that you can configure this to match the probable cause values in the receiving alarm system
tfAlarmOrigTime	the time for the first occurrence of this alarm
tfAlarmTime	the time for the last state change of this alarm
tfAlarmSeverity	the latest severity (non clear) reported for this alarm
tfAlarmCleared	boolean indicated if the latest state change reports a clear
tfAlarmText	the latest alarm text.
tfAlarmOperatorState	the latest operator alarm state such as ack
tfAlarmOperatorNote	the latest operator note

The MIB defines separate notifications for every severity level in order to support SNMP managers that only can map severity levels to individual notifications. Every notification contains the parameters of the alarm table.

SNMP Object Identifiers

Example 64. Object Identifiers

```
tfAlarmMIB             node         1.3.6.1.4.1.24961.2.103
tfAlarmObjects         node         1.3.6.1.4.1.24961.2.103.1
tfAlarms               node         1.3.6.1.4.1.24961.2.103.1.1
tfAlarmNumber          scalar       1.3.6.1.4.1.24961.2.103.1.1.1
tfAlarmLastChanged     scalar       1.3.6.1.4.1.24961.2.103.1.1.2
tfAlarmTable           table        1.3.6.1.4.1.24961.2.103.1.1.5
tfAlarmEntry           row          1.3.6.1.4.1.24961.2.103.1.1.5.1
tfAlarmIndex           column       1.3.6.1.4.1.24961.2.103.1.1.5.1.1
tfAlarmType            column       1.3.6.1.4.1.24961.2.103.1.1.5.1.2
tfAlarmDevice          column       1.3.6.1.4.1.24961.2.103.1.1.5.1.3
tfAlarmObject          column       1.3.6.1.4.1.24961.2.103.1.1.5.1.4
tfAlarmObjectOID       column       1.3.6.1.4.1.24961.2.103.1.1.5.1.5
tfAlarmObjectStr       column       1.3.6.1.4.1.24961.2.103.1.1.5.1.6
tfAlarmSpecificProblem column       1.3.6.1.4.1.24961.2.103.1.1.5.1.7
tfAlarmEventType       column       1.3.6.1.4.1.24961.2.103.1.1.5.1.8
tfAlarmProbableCause   column       1.3.6.1.4.1.24961.2.103.1.1.5.1.9
tfAlarmOrigTime        column       1.3.6.1.4.1.24961.2.103.1.1.5.1.10
tfAlarmTime            column       1.3.6.1.4.1.24961.2.103.1.1.5.1.11
tfAlarmSeverity        column       1.3.6.1.4.1.24961.2.103.1.1.5.1.12
tfAlarmCleared         column       1.3.6.1.4.1.24961.2.103.1.1.5.1.13
tfAlarmText            column       1.3.6.1.4.1.24961.2.103.1.1.5.1.14
tfAlarmOperatorState   column       1.3.6.1.4.1.24961.2.103.1.1.5.1.15
tfAlarmOperatorNote    column       1.3.6.1.4.1.24961.2.103.1.1.5.1.16
tfAlarmNotifications   node         1.3.6.1.4.1.24961.2.103.2
tfAlarmNotifsPrefix    node         1.3.6.1.4.1.24961.2.103.2.0
tfAlarmIndeterminate   notification 1.3.6.1.4.1.24961.2.103.2.0.1
tfAlarmWarning         notification 1.3.6.1.4.1.24961.2.103.2.0.2
tfAlarmMinor           notification 1.3.6.1.4.1.24961.2.103.2.0.3
tfAlarmMajor           notification 1.3.6.1.4.1.24961.2.103.2.0.4
tfAlarmCritical        notification 1.3.6.1.4.1.24961.2.103.2.0.5
tfAlarmClear           notification 1.3.6.1.4.1.24961.2.103.2.0.6
tfAlarmConformance     node         1.3.6.1.4.1.24961.2.103.10
```

```
 tfAlarmCompliances     node         1.3.6.1.4.1.24961.2.103.10.1
 tfAlarmCompliance      compliance   1.3.6.1.4.1.24961.2.103.10.1.1
 tfAlarmGroups          node         1.3.6.1.4.1.24961.2.103.10.2
 tfAlarmNotifs          group        1.3.6.1.4.1.24961.2.103.10.2.1
 tfAlarmObjs            group        1.3.6.1.4.1.24961.2.103.10.2.2
```

Using the SNMP Alarm MIB

Alarm Managers should subscribe to the notifications and read the alarm table in order to synchronize the alarm list. In order to do this you need an access view that matches the alarm MIB and create a SNMP target. Default SNMP settings in NSO lets you read the alarm MIB with v2c and community public. A target is setup in the following way, (assuming the SNMP Alarm Manager has IP address 192.168.1.1 and wants community string public in the v2c notifications):

Example 65. Subscribing to SNMP Alarms

```
$ ncs_cli -u admin -C
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# snmp notify monitor type trap tag monitor
admin@ncs(config-notify-monitor)# snmp target alarm-system ip 192.168.1.1 udp-port 162 \
        tag monitor v2c sec-name public
admin@ncs(config-target-alarm-system)# commit
Commit complete.
admin@ncs(config-target-alarm-system)# show full-configuration snmp target
snmp target alarm-system
 ip       192.168.1.1
 udp-port 162
 tag      [ monitor ]
 timeout  1500
 retries  3
 v2c sec-name public
!
snmp target monitor
 ip       127.0.0.1
 udp-port 162
 tag      [ monitor ]
 timeout  1500
 retries  3
 v2c sec-name public
!
admin@ncs(config-target-alarm-system)#
```

CHAPTER 6

NSO Alarms

Overview

NSO generates alarms for serious problems that must be remedied. Alarms are available over all north-bound interfaces and the exist at the path **/alarms**. NSO alarms are managed as any other alarms by the general NSO Alarm Manager, see the specific section on the alarm manager in order to understand the general alarm mechanisms.

The NSO alarm manager also presents a northbound SNMP view, alarms can be retrieved as an alarm table, and alarm state changes are reported as SNMP Notifications. See the "NSO Northbound" documentation on how to configure the SNMP Agent.

This is also documented in the example `/examples.ncs/getting-started/using-ncs/5-snmp-alarm-northbound`.

Alarm type structure

```
alarm-type
    ncs-cluster-alarm
        cluster-subscriber-failure
    commit-through-queue-rollback-failed
    ncs-dev-manager-alarm
        ned-live-tree-connection-failure
        dev-manager-internal-error
        final-commit-error
        commit-through-queue-blocked
        bad-user-input
        abort-error
        revision-error
        missing-transaction-id
        configuration-error
        commit-through-queue-failed
        connection-failure
        out-of-sync
    ncs-snmp-notification-receiver-alarm
        receiver-configuration-error
```

```
ncs-package-alarm
    package-load-failure
    package-operation-failure
ncs-service-manager-alarm
    service-activation-failure
```

Alarm type descriptions

- *alarm-type:* Base identity for alarm types. A unique identification of the fault, not including the managed object. Alarm types are used to identify if alarms indicate the same problem or not, for lookup into external alarm documentation, etc. Different managed object types and instances can share alarm types. If the same managed object reports the same alarm type, it is to be considered to be the same alarm. The alarm type is a simplification of the different X.733 and 3GPP alarm IRP alarm correlation mechanisms and it allows for hierarchical extensions. A 'specific-problem' can be used in addition to the alarm type in order to have different alarm types based on information not known at design-time, such as values in textual SNMP Notification varbinds.
- *ncs-cluster-alarm:* Base type for all alarms related to cluster. This is never reported, sub-identities for the specific cluster alarms are used in the alarms.
- *cluster-subscriber-failure:* Failure to establish a notification subscription towards a remote node.
- *commit-through-queue-rollback-failed:* Automatic rollback of a commit-queue item failed.
- *ncs-dev-manager-alarm:* Base type for all alarms related to the device manager This is never reported, sub-identities for the specific device alarms are used in the alarms.
- *ned-live-tree-connection-failure:* NCS failed to connect to a managed device using one of the optional live-status-protocol NEDs. Verify the configuration of the optional NEDs. If the error occurs intermittently, increase connect-timeout.
- *dev-manager-internal-error:* An internal error in NCS device manager. It might happen due to unexpected device behaviour, or other unexpected condition. Report to the support team, provide NCS error log together with other available NCS logs.
- *final-commit-error:* A managed device validated a configuration change, but failed to commit. When this happens, NCS and the device are out of sync. Reconcile by comparing and sync-from or sync-to.
- *commit-through-queue-blocked:* A commit was queued behind a queue item waiting to be able to connect to one of its devices. This is potentially dangerous since one unreachable device can potentially fill up the commit queue indefinitely.
- *bad-user-input:* Invalid input from user. NCS cannot recognize parameters needed to connect to device. Verify that the user supplied input are correct.
- *abort-error:* An error happened while aborting or reverting a transaction. Device's configuration is likely to be inconsistent with the NCS CDB. Inspect the configuration difference with compare-config, resolve conflicts with sync-from or sync-to if any.
- *revision-error:* A managed device arrived with a known module, but too new revision. Upgrade the Device NED using the new YANG revision in order to use the new features in the device.
- *missing-transaction-id:* A device announced in its NETCONF hello message that it supports the transaction-id as defined in http://tail-f.com/yang/netconf-monitoring. However when NCS tries to read the transaction-id no data is returned. The NCS check-sync feature will not work. This is usually a case of misconfigured NACM rules on the managed device.
- *configuration-error:* Invalid configuration of NCS managed device, NCS cannot recognize parameters needed to connect to device. Verify that the configuration parameters defined in tailf-ncs-devices.yang submodule are consistent for this device.
- *commit-through-queue-failed:* A queued commit failed. Resolve with rollback if possible.

- *connection-failure:* NCS failed to connect to a managed device before the timeout expired. Verify address, port, authentication, check that the device is up and running. If the error occurs intermittently, increase connect-timeout.
- *out-of-sync:* A managed device is out of sync with NCS. Usually it means that the device has been configured out of band from NCS point of view. Inspect the difference with compare-config, reconcile by invoking sync-from or sync-to.
- *ncs-snmp-notification-receiver-alarm:* The snmp-notification-receiver could not setup its configuration, either at startup or when reconfigured. Snmp notifications will now be missed. Check the error-message and change the configuration.
- *receiver-configuration-error:* An SNMP Notification Receiver configuration error prevented it from starting.
- *ncs-package-alarm:* Base type for all alarms related to packages. This is never reported, sub-identities for the specific package alarms are used in the alarms.
- *package-load-failure:* NCS failed load a package. Check the package for the reason.
- *package-operation-failure:* A package has some problem with its operation. Check the package for the reason.
- *ncs-service-manager-alarm:* Base type for all alarms related to the service manager This is never reported, sub-identities for the specific service alarms are used in the alarms.
- *service-activation-failure:* A service failed during re-deploy. Corrective action and another re-deploy is needed.