Cisco Live Amsterdam 2026

DevNet Workshop DEVWKS-1449

Lab Guide

Host: Oscar M. Cueva (omunozcu@cisco.com)

Contents

Author: Oscar M. Cueva (omunozcu@cisco.com)

Preparation Tasks

This lab hosted in Cisco dCloud and will be accessible using an eXpo URL. In order to find the link, open the following URL in a Chrome browser window:

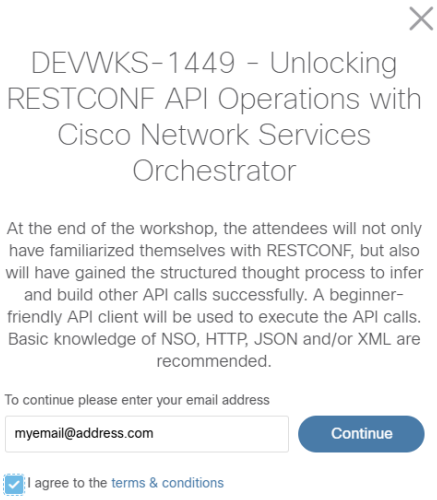
https://github.com/omunozcu/CL26AMS-DEVWKS-1449

Open the dCloud link for your session, which should look like this:

https://expo.ciscodcloud.com/<random_string>

Please click on the “Explore” button:

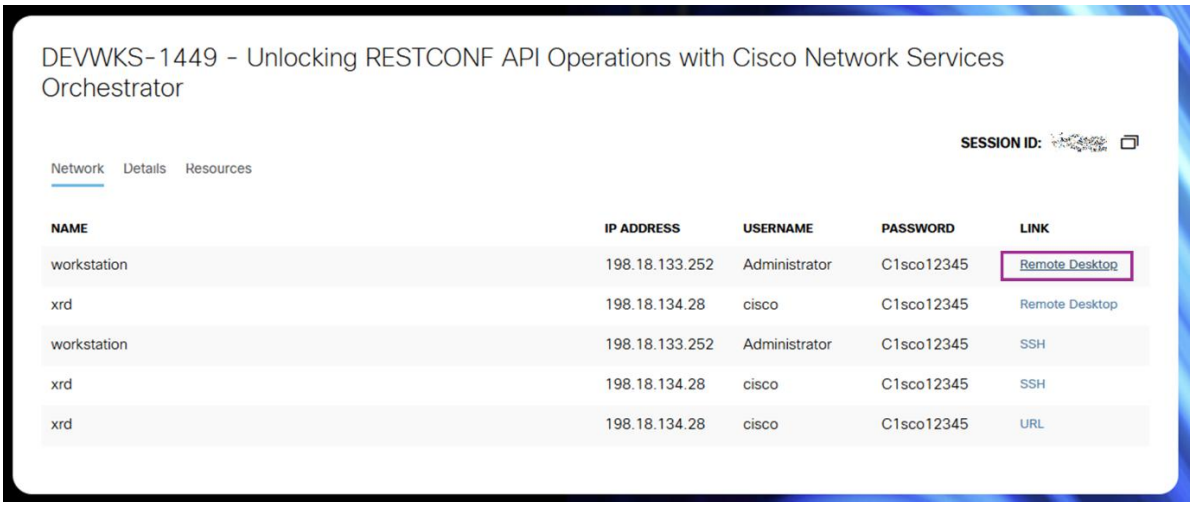
Please type your email address, check the box agreeing to the terms and conditions, and click “Continue”.



Author: Oscar M. Cueva (omunozcu@cisco.com)

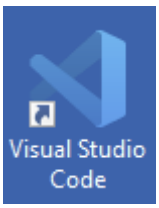
Note: you won't receive any email at the provided address.

Now, click on the workstation's "Remote Desktop":



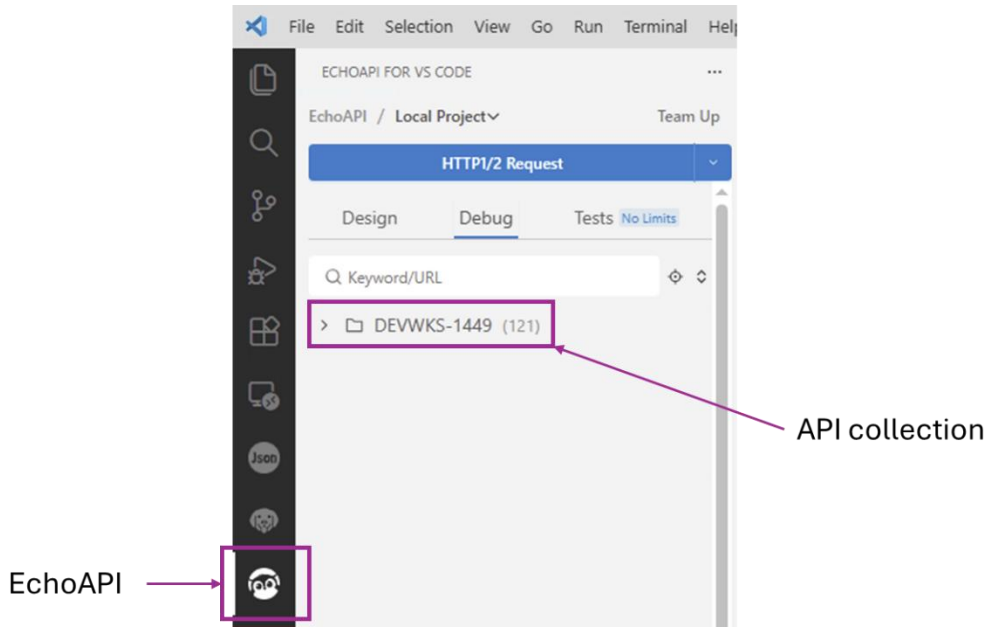
A new browser window will open, showing a Windows remote desktop. This is the screen we'll be working on. If a browser window opens in the remote desktop, just minimize it.

Please open Visual Studio Code in the remote desktop. You'll find shortcuts in the remote desktop and in the task bar:



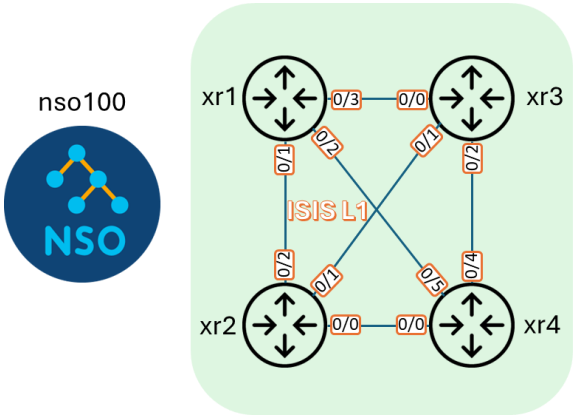
Once VS Code opens, click on the EchoAPI icon. You'll see the "DEVWKS-1449" folder, which contains all the API calls we'll be running:

Author: Oscar M. Cueva (omunozcu@cisco.com)



Lab Topology

This is the lab topology we'll be using:



It comprises:

- 4 containerized XRd instances running IOS XR version 7.11.2
- 1 containerized NSO instance running version 6.4.8.

NSO brings RESTCONF enabled by default. The RESTCONF service is available on the same port the WebUI uses, which in our setup is 8888. For further details about how to enable and tune these settings, please refer to the NSO documentation.

Author: Oscar M. Cueva (omunozcu@cisco.com)

IP addresses and credentials to connect to each device are provided in the following table:

Network Element	IP address	SSH port (CLI)	HTTP port
nso100	198.18.128.100	2024	8888
xr1	198.18.128.101	22	N/A
xr2	198.18.128.102	22	N/A
xr3	198.18.128.103	22	N/A
xr4	198.18.128.104	22	N/A

You will find shortcuts to connect to each of them via SSH terminal in the remote desktop. Alternatively, you can also use the VS Code terminal if you prefer.

Definitions

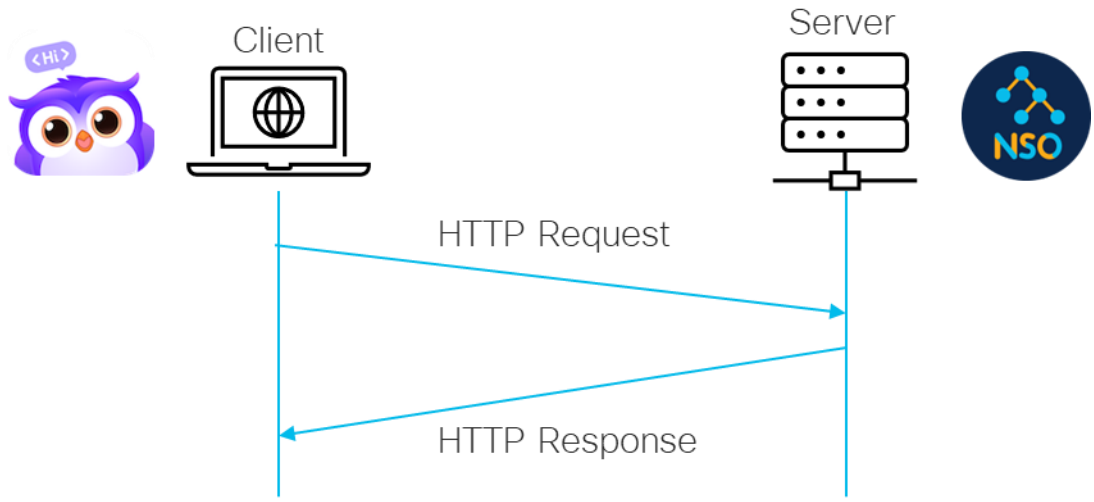
In this section we'll provide some useful definitions for those not familiar with RESTCONF. Feel free to skip to the next section if you already have some experience with REST APIs.

RESTCONF (Representational State Transfer Configuration Protocol) is a HTTP-based protocol that provides a programmatic interface for accessing and manipulating data defined in YANG. In simple terms, it's an alternative to CLI over SSH but meant to be used by machines instead of humans.

YANG (Yet Another Next Generation) is a data modeling language used to define the structure and semantics of configuration and operational data for network devices.

RESTCONF operates on a client-server model, where:

- The client sends requests to the server to manage its configuration and operational state. We will use EchoAPI (as VS Code extension) as RESTCONF client today, but many other options are available, e.g. Bruno or Postman.
- The server processes these requests to perform CRUD (Create, Read, Update, Delete) operations on YANG-modeled data. Our lab instance of Cisco Network Services Orchestrator (NSO) will act as the RESTCONF server.



During this workshop, we will learn to work with the RESTCONF client. We'll build and send API requests mainly using the EchoAPI software, this time as VS Code extension.

REST interactions are inherently stateless, i.e. each request from the client to the server contains all the necessary information, and the server does not store any client context between requests.

With RESTCONF, client requests take the form of an HTTP request, comprising four important pieces of information:

Concept	Definition
URI	Used to define the target *resource* of the request
HTTP method	Used to define the requested action on the resource
HTTP headers	Used for conveying metada about request and response, e.g. authentication and format
Payload (optional)	Actual data representing configuration, expressed either in XML or JSON format

In the context of REST APIs, a *resource* is any identifiable entity or piece of information exposed by the server to be retrieved and manipulated by the client. A resource may contain other resources, e.g. a routed interface resource in a network device contains other resources like the IP address and subnet mask, amongst others.

Regarding HTTP methods, we'll study the following ones:

HTTP Method	Purpose
GET	It reads/retrieves a resource
POST	It creates a new resource

Author: Oscar M. Cueva (omunozcu@cisco.com)

PUT	It updates/replaces an entire resource
PATCH	It updates/modifies a resource, entirely or partially
DELETE	It removes a resource

We will only work with three HTTP headers in this workshop:

HTTP Header	Definition
Authorization	Used to provide credentials information. In this workshop we'll use Basic Authorization providing the server's username and password
Accept	Used to determine the expected format of the response payload. We'll choose one of these two values: ○ application/yang-data+json to request a JSON formatted response ○ application/yang-data+xml to request an XML formatted response
Content-Type	Used to indicate the format of the request payload. They will take the same values we have seen for the Accept header

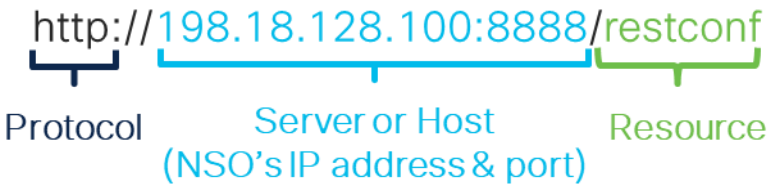
HTTP responses, on the other hand, will be analyzed based on these two pieces of information:

Concept	Definition
Status Code	Three-digit number indicating the outcome of the request, grouped in the following categories: ○ 2xx: Successful responses ○ 4xx: Client error responses ○ 5xx: Server error responses
Payload	Optional message body that carries the actual data requested by the client, or an error message

Basically, we expect to get successful responses, e.g. status codes 200 or 204. Quite often we'll see error codes, which can happen due to mistakes we have to solve, like 401 or 404, or errors at the server's side, like 500. We'll deal with all this during the workshop.

Basic Verifications

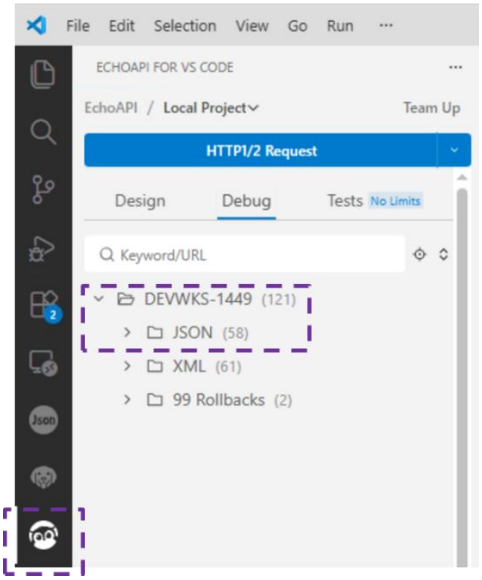
In our setup, NSO will listen to RESTCONF requests on the following URI, that we'll call *base URL* from now onwards: http://198.18.128.100:8888/restconf



Every URI we use in this workshop will contain at least the base URL in the most significant part of the string. We'll explore how to add more characters to the URI to target different resources via RESTCONF.

Time to start the hands-on activities.

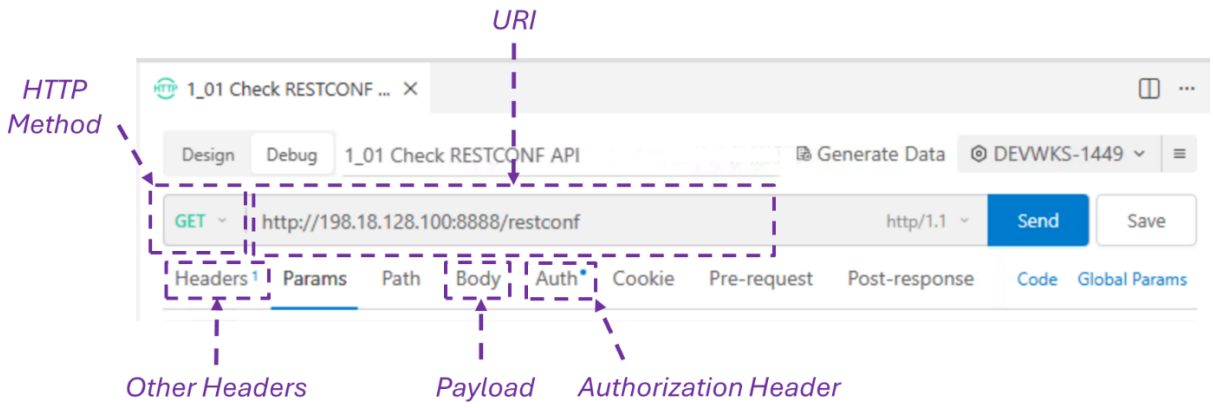
Please open VS Code and search the EchoAPI icon at the bottom left, then navigate to our collection, called DEVWKS-1449. Use the directory for the format you feel more comfortable with: XML or JSON. If you have no preference, JSON is suggested.



In this section we'll be using the first folder, called "Basic Verifications".

Please analyze the first API call named "1_01 Check RESTCONF API". Click on it and details will be shown on the right-hand side of the window. Feel free to explore the following sections:

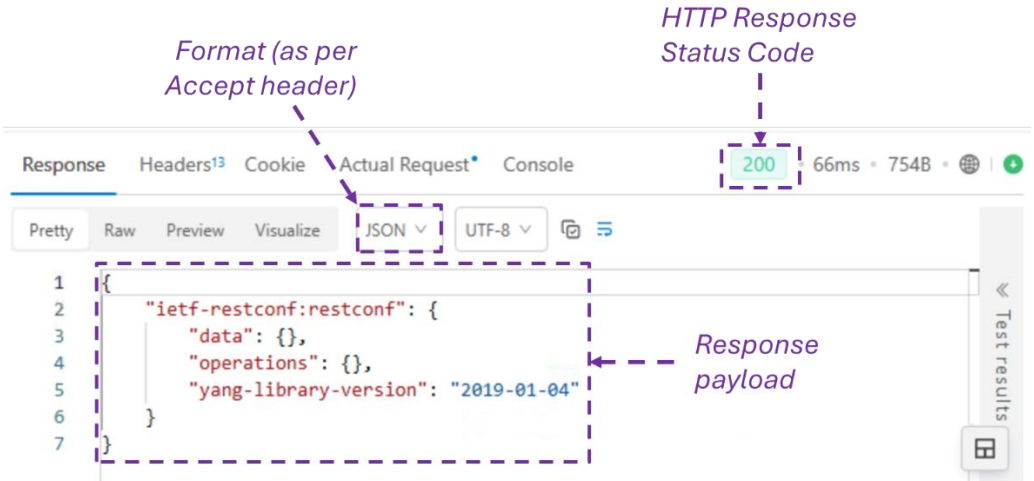
Author: Oscar M. Cueva (omunozcu@cisco.com)



Notice the HTTP method being used, the header(s), and the Auth tab.

API call name	1_01 Check RESTCONF API
HTTP Method	GET
URI	http://198.18.128.100:8888/restconf
Headers	Accept: application/yang-data+json Authorization: Basic cisco:cisco123

When you click the “Send” button, the response from NSO will show at the bottom part of the right-hand side:



Expected response:

HTTP Status Code	200 OK
Payload	{ "ietf-restconf:restconf": { "data": {},

Author: Oscar M. Cueva (omunozcu@cisco.com)

	"operations": {}, "yang-library-version": "2019-01-04" } }

Status code 200 indicates that the server could process the request correctly. The response payload informs us about the yang-library-version, as well as two available sections to explore: data and operations. We'll look into each of these shortly.

Let's explore the kind of situation we'd find if we made mistakes. The following API call is exactly like the previous one but with wrong credentials (in the Auth section). Open and run the following API call:

API call name	1_02 (Error) Check RESTCONF API Wrong Credentials
HTTP Method	GET
URI	http://198.18.128.100:8888/restconf
Headers	Accept: application/yang-data+json Authorization: Basic dummy:password

Expected response:

HTTP Status Code	401 Unauthorized
Payload	{ "ietf-restconf:errors": { "error": [{ "error-type": "protocol", "error-tag": "access-denied" }] } }

You can see the returned status code gives us a hint of the problem, and the payload tells us that our access has been denied. Remember that status codes in the 4xx range indicate an error on the client's side, so it's up to us to fix it.

From now on, we'll use the `base_url` variable in our API client to abbreviate http://198.18.128.100:8888/restconf.

Filtering Data in the Query

In this section we'll work with the API calls located in the "Query filtering" folder of our collection.

When we send a REST API query to NSO, we often expect some sort of data in the response. When querying operational or configuration data, the amount of returned information could be large.

We can always get the full response and try to find the data we need, but big queries will take long to be processed by the server, so filtering at the query level is a better approach. This improves efficiency at two levels: server processing time and bandwidth utilization on the wire.

Before we start applying filters, we can figure out the available options by running the following API call:

API call name	2_01 Query RESTCONF Capabilities
HTTP Method	GET
URI	base_url/data/ietf-restconf-monitoring:restconf-state
Headers	Accept: application/yang-data+json Authorization: Basic cisco:cisco123

The response will contain some interesting information in the payload:

HTTP Status Code	200 OK
Payload	{ "ietf-restconf-monitoring:restconf-state": { "capabilities": { "capability": ["urn:ietf:params:restconf:capability:defaults:1.0?basic-mode=explicit", "urn:ietf:params:restconf:capability:depth:1.0", "urn:ietf:params:restconf:capability:fields:1.0", "urn:ietf:params:restconf:capability:with-defaults:1.0", "urn:ietf:params:restconf:capability:filter:1.0", "urn:ietf:params:restconf:capability:replay:1.0", "urn:ietf:params:restconf:capability:yang-patch:1.0", *<omitted_output>*] } *<omitted_output>*}

Author: Oscar M. Cueva (omunozcu@cisco.com)

These are optional query parameters we can use in our URI to limit the response we get from NSO.

If you remember the response we got in the first API call we ran, titled "1_01 Check RESTCONF API", the response contained the `data` and `operations` elements.

The `data` object is the one we'll use to read from and write to NSO's Configuration Database (CDB). But the CDB contains lots of data, so querying the whole database every time is not a valid option.

For the sake of exploring the filtering possibilities, let's run the following query first:

API call name	2_02 Get RESTCONF data resource
HTTP Method	GET
URI	base_url/data
Headers	Accept: application/yang-data+json Authorization: Basic cisco:cisco123

The first thing we'll notice is that the server takes a long time to reply. Eventually, we'll get a Status Code 200 OK response with a huge payload. Some clients may even truncate the response body. Feel free to scroll down and take a look. We're basically getting a full copy of the CDB.

That's not practical at all, but sometimes we don't know where the resource we need lies in the structure… How can we explore the CDB in a lighter way? Fortunately, NSO's RESTCONF API offers several filtering options.

Let's start exploring `depth`. Run the following API call:

API call name	2_03 Get RESTCONF data resource with depth
HTTP Method	GET
URI	base_url/data?depth=1
Headers	Accept: application/yang-data+json Authorization: Basic cisco:cisco123

We've added `depth=1` as an optional parameter in the URI, which will be applied to the `data` object. The response is received much faster, and the payload is significantly lighter and easier to handle.

The `depth` query parameter controls how many levels of data are returned from a resource. With `depth=1` we get only the child nodes, but not their respective child nodes.

Feel free to adapt the value of the `depth` query parameter to 3 and run the query again to observe how the received payload increases in complexity.

The analysis of the payload received also helps us find interesting objects we want to explore further and drill down on them on subsequent requests. Run the following query as an example:

API call name	2_04 Deep-dive into data resource – devices
HTTP Method	GET
URI	base_url/ data/tailf-ncs:devices?depth=2
Headers	Accept: application/yang-data+json Authorization: Basic cisco:cisco123

Notice we've added at the end of the URI one of the child objects we got in the previous response payload (`tailf-ncs:devices`), and we're also filtering with depth.

The response looks like this:

HTTP Status Code	200 OK
Payload	{ "tailf-ncs:devices": { "global-settings": {}, "authgroups": {}, "device-group": [], "mib-group": [], "device": [], "ned-ids": {} } }

Let's continue exploring with the following query, where we have added one of the child objects received in the previous payload at the end of our URI:

API call name	2_05 Deep-dive into data resource - devices device
HTTP Method	GET
URI	base_url/ data/tailf-ncs:devices/device?depth=2
Headers	Accept: application/yang-data+json Authorization: Basic cisco:cisco123

The amount of information we receive, even limiting the result with the depth parameter, is quite large. This happens because we're querying all the configuration NSO has for all devices in the CDB.

Isn't there an easy way to select only the field(s) we're interested in? Luckily, there is. We can use the `fields` optional parameter to specify the key-value pair(s) we're interested in. Let's run the following API call to exemplify this:

API call name	2_06 Filter device names
HTTP Method	GET
URI	base_url/data/tailf-ncs:devices/device?fields=name
Headers	Accept: application/yang-data+json

Notice the received payload is limited to the `name` of each returned object, as specified by the query parameter:

HTTP Status Code	200 OK
Payload	{ "tailf-ncs:device": [{ "name": "xr1" }, { "name": "xr2" }, { "name": "xr3" }, { "name": "xr4" }] }

We can use the same query parameter to retrieve several fields, like we can observe when we run the following API call:

API call name	2_07 Filter devices and addresses option 1
HTTP Method	GET
URI	base_url/data/tailf-ncs:devices/device?fields=name;address
Headers	Accept: application/yang-data+json

Notice the received payload is limited to the `name` and `address` of each returned object:

HTTP Status Code	200 OK
Payload	{ "tailf-ncs:device": [

```
        {
            "name": "xr1",
            "address": "198.18.1.101"
        },
        {
            "name": "xr2",
            "address": "198.18.1.102"
        },
        {
            "name": "xr3",
            "address": "198.18.1.103"
        },
        {
            "name": "xr4",
            "address": "198.18.1.104"
        }
    ]
}
```


With this query we have figured out we have 4 devices, and we can address them by name in subsequent queries.

Note: depending on the client you use, this query may fail when executed requesting XML payload with the following error message: <error-message>too many instances: 4</error-message>. We will show the solution to this below.

There is an alternative way to using `fields`. So far, the filter would apply to the first level of child objects in the response, but we can also apply deep filtering in the response object tree structure, as we see in the next API call:

API call name	2_08 Filter devices and addresses option 2
HTTP Method	GET
URI	base_url/data?fields=tailf-ncs:devices/device(name;address)
Headers	Accept: application/yang-data+json

Notice the different format in the URI. Let's compare both:

- Option 1: /data/tailf-ncs:devices/device? fields=name;address
- Option 2: /data?fields=tailf-ncs:devices/device(name;address)

For those using XML, this second way of using `fields` solves the limit we found in the previous faulty query.

Author: Oscar M. Cueva (omunozcu@cisco.com)

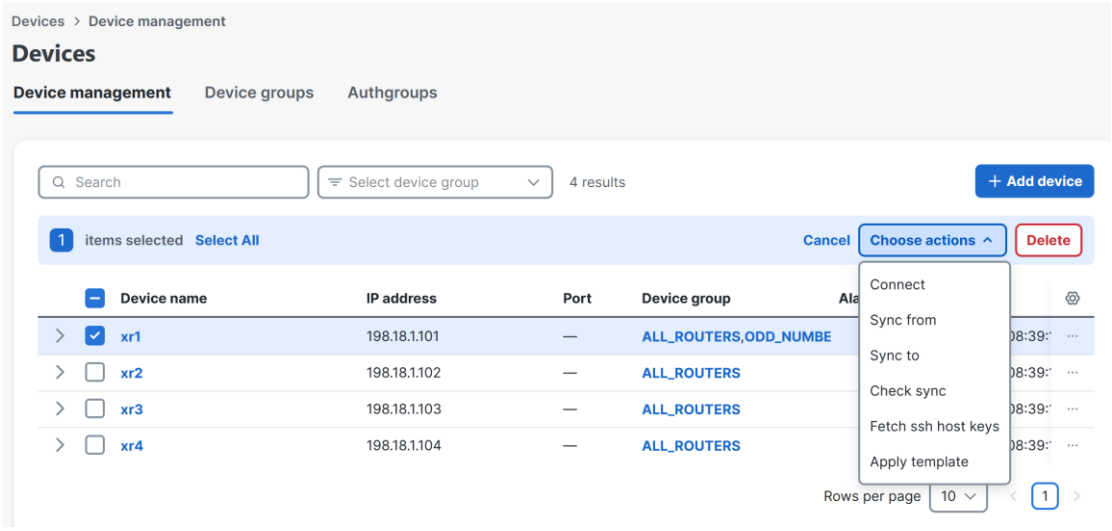
Device Actions

In this section we move to the “Device Actions” folder of our collection.

Before we start manipulating device configurations, we need to make sure NSO has connectivity to the devices, and their configurations are in sync with the CDB. We can verify this and execute different actions with the following CLI commands:

```
cisco@ncs# devices device xr1 ?
Possible completions:
<omitted_output>}
  check-sync              - Check if the NCS config is in sync with the device
  compare-config          - Compare the actual device config with the NCS copy
  connect                 - Connect to the device
  ping                    - ICMP ping the device
  ssh                     - SSH connection configuration
  sync-from               - Synchronize the config by pulling from the device
  sync-to                 - Synchronize the config by pushing to the device
<omitted_output>}
```

Likewise, the NSO WebUI also offers the option to execute such actions under the Devices menu:



Remember that when we executed our first basic verification API call, titled “1_01 Check RESTCONF API”, we got the following response:

HTTP Status Code	200 OK
Payload	{ "ietf-restconf:restconf": { "data": {}, "operations": {},

	"yang-library-version": "2019-01-04" } }

We'll be working with `operations` in this section. The logic is very simple; all API calls will look like the following example:

API call name	3_01 Device Group Fetch SSH Host Keys
HTTP Method	POST
URI	base_url/operations/tailf-ncs:devices/device-group=ALL_ROUTERS/fetch-ssh-host-keys
Headers	Accept: application/yang-data+json

We will use POST for all device operations. We still don't need to send any payload though.

If you take a look at the URI, you will notice it mirrors the CLI command we'd run to execute that action:

CLI: cisco@ncs# devices device-group ALL_ROUTERS fetch-ssh-host-keys
URI: /operations/tailf-ncs:devices/device-group=ALL_ROUTERS/fetch-ssh-host-keys

Please also notice that when we want to address a specific resource in the URI, we need to specify its key identifier with an equal sign, e.g. `device-group=ALL_ROUTERS` above.

Feel free to explore the subsequent API calls in this subfolder:

API call name	3_02 Device Group check-sync
HTTP Method	POST
URI	base_url/operations/tailf-ncs:devices/device-group=ALL_ROUTERS/check-sync
Headers	Accept: application/yang-data+json

API call name	3_03 Device sync-from
HTTP Method	POST
URI	base_url/operations/tailf-ncs:devices/device=xr1/sync-from
Headers	Accept: application/yang-data+json

API call name	3_04 Device Connect
HTTP Method	POST
URI	base_url/operations/tailf-ncs:devices/device=xr1/connect
Headers	Accept: application/yang-data+json

API call name	3_05 Device Ping
HTTP Method	POST
URI	base_url/operations/tailf-ncs:devices/device=xr1/ping
Headers	Accept: application/yang-data+json

API call name	3_06 Device check-sync
HTTP Method	POST
URI	base_url/operations/tailf-ncs:devices/device=xr1/check-sync
Headers	Accept: application/yang-data+json

API call name	3_07 Device compare-config
HTTP Method	POST
URI	base_url/operations/tailf-ncs:devices/device=xr1/compare-config
Headers	Accept: application/yang-data+json

API call name	3_08 Device sync-to
HTTP Method	POST
URI	base_url/operations/tailf-ncs:devices/device=xr1/sync-to
Headers	Accept: application/yang-data+json

You can check the URI structure resembles very much that of the equivalent NSO CLI command. For example:

CLI: cisco@ncs# devices device xr1 check-sync
URI: /operations/tailf-ncs:devices/device=xr1/check-sync

Device Configuration

We've probably arrived to the most important part of the workshop. So far, we've just queried information via RESTCONF and we've dealt with sync actions, but we haven't built any payload to describe configuration changes.

It's time to modify our devices' configurations through NSO RESTCONF API calls. We'll be describing the logic that can be applied to most RESTful APIs, and it will show that there isn't a single way to accomplish a specific goal but several, each with its own pros and cons.

Create Loopback Interface

Let's imagine I want to configure a new Loopback interface on xr1 and assign it an IPv4 address to it. This is what we do with the following API call:

API call name	4_01 (dry-run) Create Loopback 101
HTTP Method	PATCH
URI	base_url/data?dry-run=native
Headers	Accept: application/yang-data+xml Content-Type: application/yang-data+json
Payload	{ "tailf-ncs:devices": { "device": [{ "name": "xr1", "config": { "tailf-ned-cisco-ios-xr:interface": { "Loopback": [{ "id": 101, "ipv4": { "address": { "ip": "101.101.101.101", "mask": "/32" } } }] } } }] } }

Notice several things:

- The URI is quite short, we've only targeted the `/data` object after the base URL.
- We're using a new optional query parameter in the URI: `dry-run=cli`. This is the way we test whether the intended configuration is correct and NSO is able to process it before we actually instruct NSO to write the changes to the CDB, eventually affecting network devices. If you've used NSO before, you should be familiar with this feature.
- We're using a PATCH method, which intends to update resources (creating them if they don't exist).
- We've added an additional header: `Content-Type`. The value of this header will be used to inform the server (NSO in our case) of the payload format: JSON or XML.
- We've added payload to express the configuration we want to apply.

Author: Oscar M. Cueva (omunozcu@cisco.com)

The first question you may have is: how was that payload inferred? The NSO CLI is going to help us generate the payload for our API calls. After that, we can use it as a template to modify it according to our intent. Let's see how.

If we have an existing Loopback already configured on any device, we can use it as template and adapt it. Please open the NSO CLI terminal (password: cisco123) and run the following commands:

```
cisco@ncs> switch cli
cisco@ncs# show running-config devices device xr1 config interface Loopback
0
devices device xr1
 config
  interface Loopback 0
   ipv4 address 4.4.4.1 255.255.255.255
   ipv6 address 2001::1/128
   no shutdown
  exit
 !
!
```

That doesn't look like the payload we have included in our API call at all… Now try repeating the command, but adding the following:

```
cisco@ncs# show running-config devices device xr1 config interface Loopback
0 | display ?
Possible completions:
  annotations        - Display annotations
  curly-braces       - Display output as curly braces
  json               - Display output as json
  keypath            - Display output as keypath
  prefixes           - Display namespace prefixes
  restconf           - Display output as restconf path
  service-meta-data  - Display service meta information
  tags               - Display tags
  xml                - Display output as XML
  xml-template       - Display output as XML template
  xpath              - Display output as xpath
```

We see there that we have the options to show that part of the existing configuration in different formats, including JSON and XML. Let's run this command now:

```
cisco@ncs# show running-config devices device xr1 config interface Loopback
0 | display json

{
  "data": {
    "tailf-ncs:devices": {
      "device": [
        {
```

```
          "name": "xr1",
          "config": {
            "tailf-ned-cisco-ios-xr:interface": {
              "Loopback": [
                {
                  "id": 0,
                  "ipv4": {
                    "address": {
                      "ip": "4.4.4.1",
                      "mask": "255.255.255.255"
                    }
                  },
                  "ipv6": {
                    "address": {
                      "prefix-list": [
                        {
                          "prefix": "2001::1/128"
                        }
                      ]
                    }
                  }
                }
              ]
            }
          }
        }
      ]
    }
  }
}
```

Voilà! That looks very similar to the payload of our API call.

If we choose JSON, we must take the piece of code excluding the “data” keyword (highlighted in yellow above). The reason for this is that we have already included `/data` in our URI, so we’re interested in creating a resource under the `/data` object hierarchy.

If we choose XML, instead of “data”, we’ll see `<config>`, which is the part we need to exclude to build the payload for our API call:

```
cisco@ncs# show running-config devices device xr1 config interface Loopback
0 | display xml

<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>xr1</name>
      <config>
        <interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
          <Loopback>
            <id>0</id>
```

```
            <ipv4>
              <address>
                <ip>4.4.4.1</ip>
                <mask>255.255.255.255</mask>
              </address>
            </ipv4>
            <ipv6>
              <address>
                <prefix-list>
                  <prefix>2001::1/128</prefix>
                </prefix-list>
              </address>
            </ipv6>
          </Loopback>
        </interface>
      </config>
    </device>
  </devices>
</config>
```

We can use that code as baseline to create new Loopback interfaces, adapting the id, address ip, mask, etc, and discarding the fields we don't need, like the IPv6 address in this example.

Great, but what if we have no existing configuration to use as a template? There is another way to generate it for any new piece of configuration in NSO CLI. Please run the following commands:

```
cisco@ncs# config
Entering configuration mode terminal
cisco@ncs(config)# devices device xr1 config interface Loopback 101 ipv4
address 101.101.101.101 /32
cisco@ncs(config-if)# show config
interface Loopback 101
 ipv4 address 101.101.101.101 /32
 no shutdown
exit
```

Basically, we have typed the commands to create the desired configuration, but we haven't committed it.

There are two commands that we can use to generate the XML payload for our API call:

```
cisco@ncs(config-if)# show configuration | display xml
<devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>xr1</name>
    <config>
      <interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
        <Loopback>
```

```
          <id>101</id>
          <ipv4>
            <address>
              <ip>101.101.101.101</ip>
              <mask>/32</mask>
            </address>
          </ipv4>
        </Loopback>
      </interface>
    </config>
  </device>
</devices>
```

```
cisco@ncs(config-if)# commit dry-run outformat xml
result-xml {
    local-node {
        data <devices xmlns="http://tail-f.com/ns/ncs">
               <device>
                 <name>xr1</name>
                 <config>
                   <interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
                     <Loopback>
                       <id>101</id>
                       <ipv4>
                         <address>
                           <ip>101.101.101.101</ip>
                           <mask>/32</mask>
                         </address>
                       </ipv4>
                     </Loopback>
                   </interface>
                 </config>
               </device>
             </devices>
    }
}
```

Unfortunately, there's no JSON version for these two commands, so we could either use XML for our API call payload, by adapting the 'Content-Type' header accordingly, or turn that XML code to JSON format.

Let's close the configuration mode of NSO CLI without committing the changes before continuing:

```
cisco@ncs(config-if)# end
Uncommitted changes found, commit them? [yes/no/CANCEL] no
cisco@ncs#
```

Now that we understand the logic, please run the API call titled "4_01 (dry-run) Create Loopback 101".

We're using a PATCH HTTP method and we're targeting the simplest URI option, just the `/data` object. This is a very common approach. In the payload we must include the whole object structure under `/data`, adapting the Loopback id, IP address and mask to the desired values.

Please observe the positive response from the NSO server, although the configuration has not been written to the device because we used the `dry-run` option.

Now, let's show an alternative way to accomplish the same result but using POST instead of PATCH. Feel free to analyze and run the following API call:

API call name	4_02 (dry-run) Create Loopback 101
HTTP Method	POST
URI	base_url/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-xr:interface?dry-run=native
Headers	Accept: application/yang-data+xml Content-Type: application/yang-data+json
Payload	{ "Loopback": { "id": 101, "ipv4": { "address": { "ip": "101.101.101.101", "mask": "/32" } } } }

POST is used to create non-existing objects, so we need to make sure the URI targets only the intended parent resource, and the payload describes a non-existing object, else it won't work. In this case we want to create a new interface object, so the payload starts with the parent object we want to create, i.e. the Loopback interface, with all the resources under it (id, ip address, etc).

But how can we make sure we use the right URI? Again, NSO CLI comes to our rescue. Run the following command:

```
cisco@ncs# show running-config devices device xr1 config interface Loopback
0 | display restconf

/restconf/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-
xr:interface/Loopback=0/ipv4/address/ip 4.4.4.1
/restconf/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-
xr:interface/Loopback=0/ipv4/address/mask 255.255.255.255
```

Author: Oscar M. Cueva (omunozcu@cisco.com)

```
/restconf/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-xr:interface/Loopback=0/ipv6/address/prefix-list=2001%3A%3A1%2F128
```

The `display restconf` option shows us the paths we can use to address any existing resource with RESTCONF, using the corresponding HTTP method. The current values of the existing resources are shown, but in this case we can ignore that part.

Feel free to create the Loopback interface now with the "Create Loopback 101" API call, either using the PATCH (4_03) or POST (4_04) option; both have the same effect.

We should get a 200 OK response code. Additionally, notice we've added `?rollback-id=true` at the end of the URI. This is another optional parameter that triggers the return of a rollback-id as part of the response payload. If we wanted to undo the configuration, we would execute the API call contained in the "99 Rollbacks" folder. This is not mandatory but it's good practice because it enables us to quickly roll the applied configuration back when necessary.

In order to verify it worked, you can connect to the xr1 router and check that the interface indeed exists now, as well as the configuration that has been sent to the device by NSO:

```
RP/0/RP0/CPU0:xr-1#show ip interface brief

Interface                  IP-Address      Status          Protocol Vrf-Name
Loopback0                  4.4.4.1         Up              Up       default
Loopback101                101.101.101.101 Up              Up       default
MgmtEth0/RP0/CPU0/0        198.18.1.101    Up              Up       default
GigabitEthernet0/0/0/1     10.1.2.1        Up              Up       default
GigabitEthernet0/0/0/2     10.1.4.1        Up              Up       default
GigabitEthernet0/0/0/3     10.1.3.1        Up              Up       default
GigabitEthernet0/0/0/4     unassigned      Shutdown        Down     default
GigabitEthernet0/0/0/5     unassigned      Shutdown        Down     default

RP/0/RP0/CPU0:xr-1#show configuration commit changes last 1
!! Building configuration...
!! IOS XR Configuration 7.11.2
interface Loopback101
 ipv4 address 101.101.101.101 255.255.255.255
!
end
```

With this example, we've seen that we can use both POST and PATCH methods to create new resources, although the URI and payload are different in each case. NSO takes the request, turns it into commands the affected devices understand, and applies the necessary configuration changes on the targeted devices to move from the current to the desired state.

Author: Oscar M. Cueva (omunozcu@cisco.com)

There is another interesting test we can run. Let's run again the POST API call "4_04 Create Loopback 101" and see what happens.

Since we're instructing NSO to create a resource that already exists, the returned status code will be 409 Conflict, together with an error message in the payload indicating "object already exists".

If we try to create the (now existing) resource with the PATCH "Create Loopback 101" API call instead, the server won't return any error. We'll get a response code 204 No Content, indicating that the desired state has been accomplished, although no action has been taken.

These examples prove the idempotency of the RESTCONF API calls. Executing the same API call several times may generate error codes, but the resulting state won't vary, because the intent remains the same.

PATCH has one additional advantage when it comes to resource creation or modification, and it's the possibility to affect many resources with a single request. Analyze the following API call:

API call name	4_05 (dry-run) Create Loopbacks in two devices
HTTP Method	PATCH
URI	base_url/data?dry-run=native
Headers	Accept: application/yang-data+xml Content-Type: application/yang-data+json
Payload	{ "tailf-ncs:devices": { "device": [{ "name": "xr1", "config": { "tailf-ned-cisco-ios-xr:interface": { "Loopback": [{ "id": 111, "ipv4": { "address": { "ip": "111.111.111.111", "mask": "/32" } } }]

Author: Oscar M. Cueva (omunozcu@cisco.com)

```
                }
            }
        },
        {
            "name": "xr2",
            "config": {
                "tailf-ned-cisco-ios-xr:interface": {
                    "Loopback": [
                        {
                            "id": 112,
                            "ipv4": {
                                "address": {
                                    "ip": "112.112.112.112",
                                    "mask": "/32"
                                }
                            }
                        }
                    ]
                }
            }
        }
    ]
  }
}
```

Notice in the payload that we're including two separate pieces of configuration, one Loopback interface for device xr1 and another one for device xr2. Feel free to run the API call and observe the positive response from the NSO server.

After that, please run the API call "Create Loopbacks in two devices" to confirm the change and feel free to verify via CLI that the changes did indeed take effect in the intended devices.

We've learned that by targeting `/data` with a PATCH method, we could modify different configurations in different parts of the resource structure, even affecting multiple devices. This is something we could not do with POST or PUT, since the URI has to be much more specific than with PATCH, targeting single resources.

Edit Loopback

We're going to see different ways to modify and delete existing resources now, using the Loopback interfaces we just created as examples. Please move to the corresponding folder in our collection and run the following API call:

API call name	5_01 Get Loopback 101
HTTP Method	GET
URI	base_url/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-xr:interface/Loopback=101
Headers	Accept: application/yang-data+xml

Notice in the URI that we're retrieving a specific resource object: the interface Loopback101 of device xr1. We can use that URI and the payload we receive as baseline to build subsequent API calls.

The API following API call is designed to modify the IPv4 address of that Loopback:

API call name	5_02 (dry-run) Edit Loopback 101 IPv4 address (general)
HTTP Method	PATCH
URI	base_url/data?dry-run=cli
Headers	Accept: application/yang-data+xml Content-Type: application/yang-data+json
Payload	{ "tailf-ncs:devices": { "device": [{ "name": "xr1", "config": { "tailf-ned-cisco-ios-xr:interface": { "Loopback": [{ "id": 101, "ipv4": { "address": { "ip": "221.221.221.221" } } }] } } }] } }

Author: Oscar M. Cueva (omunozcu@cisco.com)

An alternative to accomplishing the same objective, also using PATCH, is shown in the following API call:

API call name	5_03 (dry-run) Edit Loopback 101 IPv4 address (specific)
HTTP Method	PATCH
URI	base_url/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-xr:interface/Loopback=101/ipv4/address/ip?dry-run=cli
Headers	Accept: application/yang-data+xml Content-Type: application/yang-data+json
Payload	{ "ip": "221.221.221.221" }

Feel free to run it. Notice we're being as specific as it gets in the URI, therefore reducing the amount of necessary payload very much. We're doing this with a PATCH method (used to modify information).

The main difference between both approaches is that in 5_02 the we're using a much shorter URI than in 5_03, which forces us to add much more information as part of the payload. You can execute both and observe the response payloads, which indicate that the IP address (and only the IP address) will be replaced. They would produce the same result.

Haven't you noticed something strange? It seems we can successfully modify an interface IP address without even mentioning the subnet mask… In fact, this is something that works with a PATCH method, because the object already exists and we are replacing the old IPv4 address value with a new one, which means that the existing subnet mask remains unmodified. That's the reason we don't need to explicitly indicate the new value.

It wouldn't work with a different HTTP method than PATCH though. Let's prove that by attempting to modify the same IP address using PUT:

API call name	5_04 (Error) Replace Loopback 101 IPv4 address
HTTP Method	PUT
URI	base_url/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-xr:interface/Loopback=101/ipv4/address?rollback-id=true
Headers	Accept: application/yang-data+xml Content-Type: application/yang-data+json
Payload	{ "address": { "ip": "221.221.221.221" } }

It's the very same request we ran before (in dry-run mode), just replacing PATCH with PUT. Please execute the API call if you haven't done so yet.

In the response, we get a 500 Internal Server Error. Remember error codes in the range of 5xx represent server-side errors. Basically, the syntax of the API request was correct, but NSO can't execute the requested action.

The reason it fails is that PUT replaces the entire object targeted by the URI, the ip address object in this case, but it's not possible to define a new IP address without a subnet mask. Even if NSO tried to push this configuration to the device, it'd be rejected. Please notice PUT refers to the resource in the URI (`ipv4/address` in this case), which will be replaced entirely by the value in the payload. Even if the old value did have a subnet mask, the use of PUT will remove it, forcing us to assign a value for the new object.

In fact, the error message we get in the response payload gives us a hint of what's happening:

```
"External error in the NED implementation for device xr1: config in diff
couldn't be parsed:\n  skipped 1 line in context '/interface/Loopback'
:\n   (line 2) : ' ipv4 address 221.221.221.221'\nNOTE: this might indicate
invalid usage of device model, set config on device and sync-from to verify
config syntax/structure".
```

Basically, the NED can't apply the requested configuration. Does it mean we can't use PUT to modify the IP address then? Not at all. We can use PUT but we need to attach the right payload to it, as we do in the next API call:

API call name	5_05 (dry-run) Replace Loopback 201 IPv4 address
HTTP Method	PUT
URI	base_url/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-xr:interface/Loopback=101/ipv4/address?rollback-id=true
Headers	Accept: application/yang-data+xml Content-Type: application/yang-data+json
Payload	{ "address": { "ip": "11.11.11.11", "mask": "/32" } }

Please send the API call and observe the positive response now.

Now that we understand our options, pick one of the possible valid options and execute the change with either of the API calls:

- 5_06 Edit Loopback 101 IPv4 address (PATCH), or
- 5_07 Replace Loopback 101 IPv4 address" (PUT)

After running it, you may verify at the router configuration that the corresponding IP address has indeed been updated.

We've seen how to edit pieces of information in the devices. How about removing them? The same logic we used to infer the right URI for the target resource is used, with the difference that we'll use a DELETE HTTP method now, as we can see by running the last API call of this folder:

API call name	5_08 Remove Loopback 101 xr1
HTTP Method	DELETE
URI	base_url/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-xr:interface/Loopback=101?rollback-id=true
Headers	Accept: application/yang-data+json

Applying Interface Shutdown

We're slowly heading towards the end of the workshop.

We'll be working with the "6 Interface Shutdown" folder in our collection now. Let's look at something different now. We have seen how to configure and modify pieces of configuration that require a value, like IP addresses or subnet masks. But what if no value is required? How do we work with features, options and states?

Let's explore this case with an example. Please run the following command on router xr1:

```
RP/0/RP0/CPU0:xr-1#show running-config interface GigabitEthernet 0/0/0/1

interface GigabitEthernet0/0/0/1
 description to xr-2-Gi0/0/0/2
 cdp
 ipv4 address 10.1.2.1 255.255.255.0
 ipv6 enable
!
```

Notice two interesting fields: `cdp` and `ipv6 enable`. This means we have enabled cdp and ipv6 on that interface, but no config settings are needed to do that… Furthermore, there is an "invisible" command we can't even see there. Can you guess? It's the `no shutdown` command somebody had to type on it in order to bring the interface up.

Let's take a different interface on the same router to exemplify this:

```
RP/0/RP0/CPU0:xr-1#show running-config interface GigabitEthernet 0/0/0/4

interface GigabitEthernet0/0/0/4
 shutdown
!
```

How do we deal with that type of configuration settings that require no value? How do we add them when they're missing or make them go away when they're present?

First, let's explore how NSO stores those pieces of configuration in the CDB:

In JSON:

```
cisco@ncs# show running-config devices device xr1 config interface
GigabitEthernet 0/0/0/1 | display json
{
  "data": {
    "tailf-ncs:devices": {
      "device": [
        {
          "name": "xr1",
          "config": {
            "tailf-ned-cisco-ios-xr:interface": {
              "GigabitEthernet": [
                {
                  "id": "0/0/0/1",
                  "description": "to xr-2-Gi0/0/0/2",
                  "ipv4": {
                    "address": {
                      "ip": "10.1.2.1",
                      "mask": "255.255.255.0"
                    }
                  },
                  "ipv6": {
                    "enable": [null]
                  },
                  "cdp": [null]
                }
              ]
            }
          }
        }
      ]
    }
  }
}
```

Author: Oscar M. Cueva (omunozcu@cisco.com)

```
cisco@ncs# show running-config devices device xr1 config interface
GigabitEthernet 0/0/0/4 | display json
{
  "data": {
    "tailf-ncs:devices": {
      "device": [
        {
          "name": "xr1",
          "config": {
            "tailf-ned-cisco-ios-xr:interface": {
              "GigabitEthernet": [
                {
                  "id": "0/0/0/4",
                  "shutdown": [null]
                }
              ]
            }
          }
        }
      ]
    }
  }
}
```

And in XML:

```
cisco@ncs# show running-config devices device xr1 config interface
GigabitEthernet 0/0/0/1 | display xml
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>xr1</name>
      <config>
        <interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
          <GigabitEthernet>
            <id>0/0/0/1</id>
            <description>to xr-2-Gi0/0/0/2</description>
            <ipv4>
              <address>
                <ip>10.1.2.1</ip>
                <mask>255.255.255.0</mask>
              </address>
            </ipv4>
            <ipv6>
              <enable/>
            </ipv6>
            <cdp/>
          </GigabitEthernet>
        </interface>
      </config>
    </device>
  </devices>
</config>
```

Author: Oscar M. Cueva (omunozcu@cisco.com)

```
cisco@ncs# show running-config devices device xr1 config interface
GigabitEthernet 0/0/0/4 | display xml
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <device>
      <name>xr1</name>
      <config>
        <interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
          <GigabitEthernet>
            <id>0/0/0/4</id>
            <shutdown/>
          </GigabitEthernet>
        </interface>
      </config>
    </device>
  </devices>
</config>
```

That's new, right? This basically means that when we want to apply the "shutdown" command to an interface, or any similar configuration, we need to add the following payload to the corresponding resource using POST, PUT or PATCH (considering their corresponding nuances).

In JSON:

```
{
  "shutdown": [null],
  "cdp": [null]
}
```

In XML:

```
<shutdown/>
<cdp/>
```

Likewise, if we want to get rid of them, we would apply a DELETE method to the corresponding resource object.

Let's experiment with these options using shutdown as an example.

Observe and run the following API call:

API call name	6_01 (dry-run) Shutdown interface option 1
HTTP Method	PATCH
URI	base_url/data?dry-run=cli
Headers	Accept: application/yang-data+xml Content-Type: application/yang-data+json

Payload	{ "tailf-ncs:devices": { "device": [{ "name": "xr1", "config": { "tailf-ned-cisco-ios-xr:interface": { "GigabitEthernet": [{ "id": "0/0/0/1", "shutdown": [null] }] } } }] } }

Observe in the response payload the command that would be added to the device configuration on the corresponding interface.

As usual, let's offer an alternative way to accomplish the same using POST:

API call name	6_02 (dry-run) Shutdown interface option 2
HTTP Method	POST
URI	base_url/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-xr:interface/GigabitEthernet=0%2F0%2F0%2F1?dry-run=cli
Headers	Accept: application/yang-data+xml Content-Type: application/yang-data+json
Payload	{ "shutdown": [null] }

POST demands a very specific URI targeting only the resource to be created, in this case the "shutdown" object with its corresponding null value.

You may have noticed some strange characters in the URI. This happens because the interface id is 0/0/0/1, but the "/" character can't be used in the URI as part of a string. The URI escape code we need to use to replace that character is "%2F".

Let's go ahead and shut the interface down with the "6_03 Shutdown interface" API call.

If you feel like it, you can verify in the xr1 device configuration that interface GigabitEthernet 0/0/0/1 is indeed in shutdown status now.

In order to bring the interface back up, all we need to do is to remove the “shutdown” object with a DELETE method, as we can observe by running the following API call:

API call name	6_04 (dry-run) Remove shutdown interface
HTTP Method	DELETE
URI	base_url/data/tailf-ncs:devices/device=xr1/config/tailf-ned-cisco-ios-xr:interface/GigabitEthernet=0%2F0%2F0%2F1/shutdown?dry-run=cli
Headers	Accept: application/yang-data+xml

You can take a look at the response payload and see the “shutdown” object has a minus sign, which means it’ll be removed from the configuration with that REST API call. Please go ahead and execute the “Remove shutdown interface” API call to execute this config change.

Now as a curiosity, if you try to execute the same API call twice, the second time (and all successive ones) you’ll get an error because the “shutdown” object no longer exists.

Is that all there is to this topic? Not really. If you have been working with JSON, you’ll notice there is no more API calls in this folder, but actually we have one alternative option to run these actions when we work with XML.

In order to explore this additional way to remove objects from the configuration, let’s use a method we have already played with in this workshop. Please connect to NSO CLI and simulate the removal of the “shutdown” object of an active interface on a device, for example:

```
cisco@ncs# config
Entering configuration mode terminal
cisco@ncs(config)# devices device xr1 config interface GigabitEthernet
0/0/0/4
cisco@ncs(config-if)# no shutdown
cisco@ncs(config-if)# show config
interface GigabitEthernet 0/0/0/4
 no shutdown
exit
```

We have typed the configuration commands on NSO to bring a device interface up, without committing them. Nothing spectacular… but let’s try to generate the corresponding XML payload for that configuration change:

```
cisco@ncs(config-if)# show config | display xml
<devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>xr1</name>
    <config>
      <interface xmlns="http://tail-f.com/ned/cisco-ios-xr">
        <GigabitEthernet>
          <id>0/0/0/4</id>
          <shutdown xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
                    nc:operation="delete"/>
        </GigabitEthernet>
      </interface>
    </config>
  </device>
</devices>
```

By the way, you may have used the command `commit dry-run outformat` xml to generate that code too, both are valid options.

The interesting part is that the `<shutdown/>` object in the XML payload now contains some extra code: `nc:operation="delete"`.

First, let's end the config mode of NSO without committing the change:

```
cisco@ncs(config-if)# end
Uncommitted changes found, commit them? [yes/no/CANCEL] no
```

Now, let's move back to our API client and run our API call:

API call name	6_05 (dry-run) Remove shutdown interface option 2
HTTP Method	PATCH
URI	base_url/data?dry-run=cli
Headers	Accept: application/yang-data+xml Content-Type: application/yang-data+xml
Payload	<devices xmlns="http://tail-f.com/ns/ncs"> <device> <name>xr1</name> <config> <interface xmlns="http://tail-f.com/ned/cisco-ios-xr"> <GigabitEthernet> <id>0/0/0/4</id> <shutdown xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation="delete"/> </GigabitEthernet> </interface> </config>

	</device> </devices>

First, notice we're using a PATCH method to delete an object. Interesting, right? This is possible because the removal of an object is a sort of modification too.

Please observe the positive HTTP response code and the matching payload confirming that NSO has understood this request and would remove the "shutdown" object from the interface. We could use this alternative method to deal with `cdp`, `ipv6 enable` and similar pieces of configuration.

What is the advantage of this second method then? As we already mentioned, with PATCH we can target several pieces on configuration, even affecting multiple devices, within a single API call. DELETE targets one and only one resource, but PATCH, given that it accepts a more generic URI, can be used to target multiple parts of the configuration in an atomic way as part of a single transaction. This is not only more efficient, but it can be beneficial in some situations because the transaction would either succeed or fail as a whole.

Feel free to explore and run either of the API calls 6_06 and 6_07.

Conclusion

This workshop ends here. We hope you have found it useful and are in a better position to start or continue using APIs effectively by leveraging the acquired knowledge about HTTP methods, building the right URI and payload to accomplish your goal.

Please remember to always go through the API documentation of the system you will be using to get the right usage guidelines. In the case of NSO, you can find it in the following link:

https://nso-docs.cisco.com/guides/development/core-concepts/northbound-apis/restconf-api

In order to continue your programmability journey, we invite you to explore the "Code" button you'll find on each of the API calls you've run. It'll show you the equivalent code for several programming languages. This way, you can use clients like EchoAPI, Bruno or Postman for development, and turn a sequence of API calls into a program or script, adding extra code between API calls to process the payload.

Feel free to contact me at omunozcu@cisco.com if you have any further questions.