

**Model Driven Network  
Automation with IOS-XE  
LTRCRT-2700**

**Speakers: Tony Roman – Content Engineer**

## Table of Contents

<b>LAB 1: Exploring RESTCONF on IOS-EX with Postman .....</b>	<b>3</b>
Introduction .....	3
Task 1: Getting Familiar with Postman .....	3
Task 2: Retrieving Device Configurations .....	5
Task 3: Making a Configuration Change .....	12
<b>Lab 2: Automating IOS-XE with RESTCONF while using Python requests.....</b>	<b>19</b>
Task 1: Getting Familiar with Python Requests .....	19
Task 2: Making a GET request with Python .....	22
Task 3: Retrieve Interface Configurations.....	24
Task 4: Get Dynamic Routing Configuration .....	27
<b>Challenge 1: Update Interface GigabitEthernet2.....</b>	<b>30</b>
Challenge 1: Solution.....	33
<b>Challenge 2: Create a Loopback Interface.....</b>	<b>34</b>
Challenge 2: Solution.....	36
<b>Lab 3: Automating IOS-XE with NETCONF using Python ncclient.....</b>	<b>37</b>
Task 1: Getting Familiar with Python ncclient .....	37
Task 2: Retrieving the Running Configuration .....	41
Task 3 - Retrieving Interfaces .....	42
Task 4 - Configuring an Interface (Default operation is merge) .....	44
Task 5 - Configuring an Interface using Replace Operation.....	46
Task 6 - Retrieving all dynamic routing configuration.....	48
<b>Challenge 3: Replace the Complete routing configuration.....</b>	<b>49</b>
Challenge 3: Solution.....	51

# LAB 1: Exploring RESTCONF on IOS-EX with Postman

## Introduction

In this lab, you will get familiar with a popular REST Client called Postman. Postman is a Chrome application that makes it extremely easy and intuitive for exploring, testing, and prototyping HTTP-Based APIs.

Using Postman, you're able to explore APIs, get to understand and test them before writing any code speeding up the overall development process.

## Task 1: Getting Familiar with Postman

In this first task, we're going to go over the basics of working with Postman and HTTP-Based APIs. First off, you should know you need to be aware of a few things when making RESTful HTTP-Based API calls. For example, you need to know:

The URL of the API resource in question – in networking, this usually maps back to a given feature, statistics, configuration, or operational state data.

The HTTP method (verb) required – common ones are GET, PUT, POST, PATCH, and DELETE.

Headers – used to define attributes such as encoding types, i.e. do you want to use XML or JSON to communicate to the server (network device)—remember the network device is now in essence running a web server)? The most common headers we use in the labs are **Content-Type** and **Accept**. **Accept** tells the target device how you want to receive data and **Content-Type** is how you are sending data. Note: you send data when you are making a configuration change, but do not when retrieving data.

Credentials – of course, you need some level of security or credentials. For IOS-XE, you require level 15 credentials to use the API.

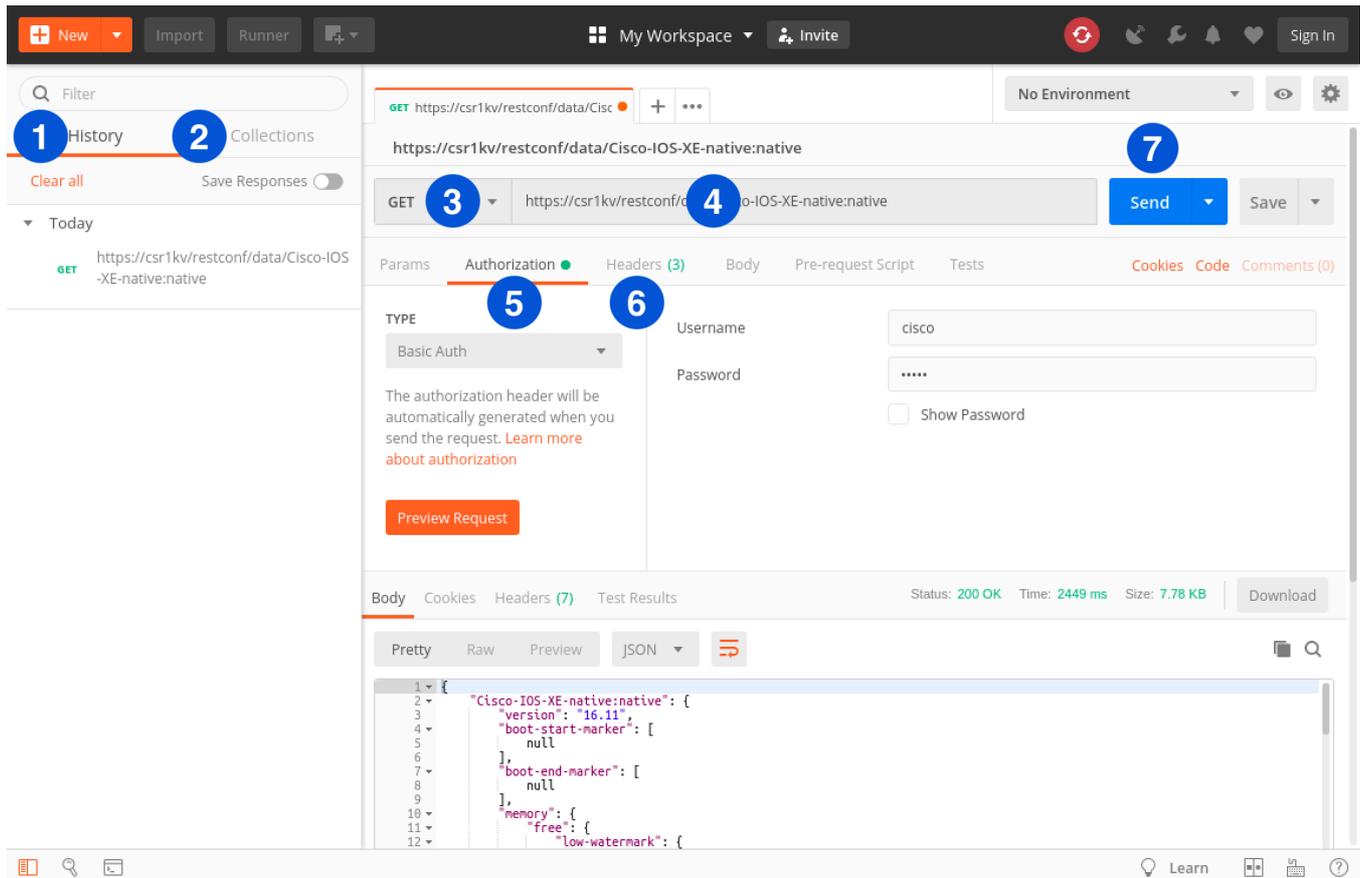
That's enough for now—let's get started.

### Step 1

Connect to your assigned lab pod and open the Student Workstation. The Student Workstation is a pre-built Ubuntu Linux instance that has utilities like Postman pre-installed. Once you have accessed the Student Workstation desktop, open Postman by clicking the icon located on the desktop.

## Step 2

Once you have opened Postman, you'll see a screen similar to the following one:



Let's take a minute to review all the numbers in the picture. Each number identifies a key area of Postman that's used to make API calls. Here is a breakdown of what each number is:

1. **History** – every API call you make is saved in History. Since Postman is a Chrome application, History is shared between Postman (Chrome) sessions if you're logged in different PCs.
2. **Collections** – you can create a collection of items from your History. This makes it possible to build out robust workflows and save them as a collection before you start development / coding.
3. **HTTP Verb / Method** – drop down for you to select the method required for the API: GET, POST, PATCH, PUT, DELETE, etc.
4. **URL** – this is where you enter the URL of your API endpoint.
5. **Authorization** – your level 15 credentials will go here for IOS-XE (we'll be using Basic Authentication).
6. **Headers** – you will configure two headers here, **Content-Type** and **Accept** that'll get sent in the HTTP Header as part of the request.

Finally, the **Send** button that issues the API call to the device (API endpoint)

## Task 2: Retrieving Device Configurations

### Step 1

It's time to start issuing API calls to the Cisco IOS-XE CSR1000V.

---

*Note: This device is running IOS XE 16.11.1b*

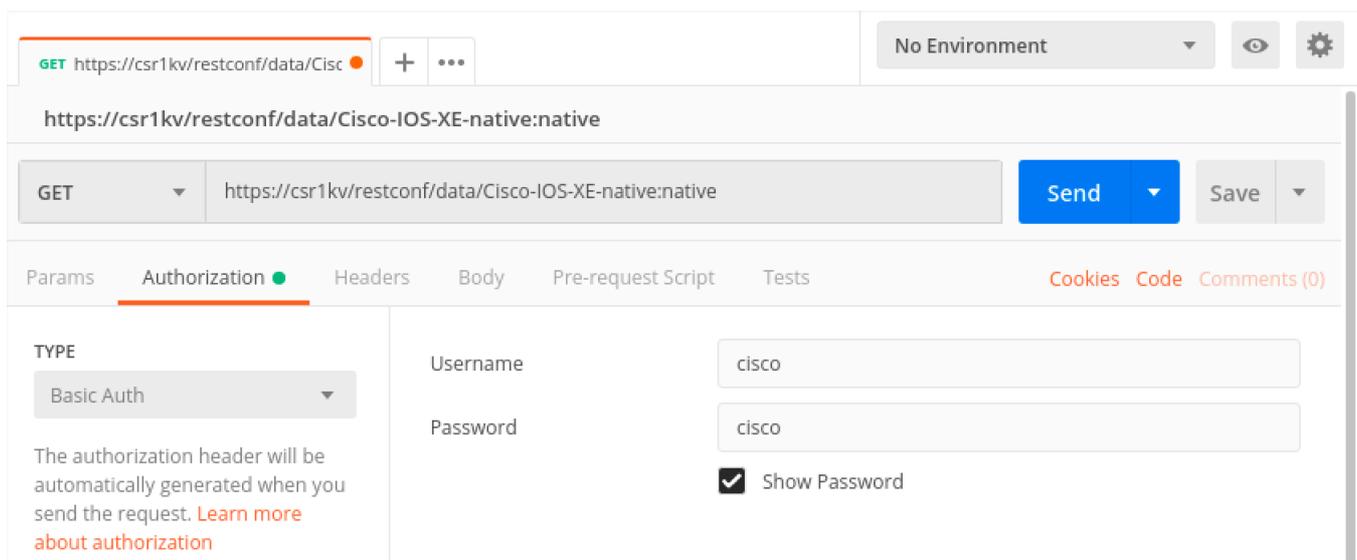
---

The first API call you'll issue will retrieve a complete device running configuration as **JSON** as compared to a "show run" that is raw text and has no structure to it at all.

Enter in the following URL: `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native`

Ensure the HTTP method is *GET*

Configure your credentials (*cisco/cisco*) in the Authorization tab



The screenshot shows a REST client interface with the following elements:

- Request Method: GET
- URL: `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native`
- Buttons: Send, Save
- Authorization Tab: Selected, showing Basic Auth configuration.
- Username: `cisco`
- Password: `cisco`
- Show Password:
- Additional tabs: Params, Headers, Body, Pre-request Script, Tests, Cookies, Code, Comments (0)

Here we are getting back the "running" configuration as denoted in the URL that adheres to the Cisco "native" **YANG** model.

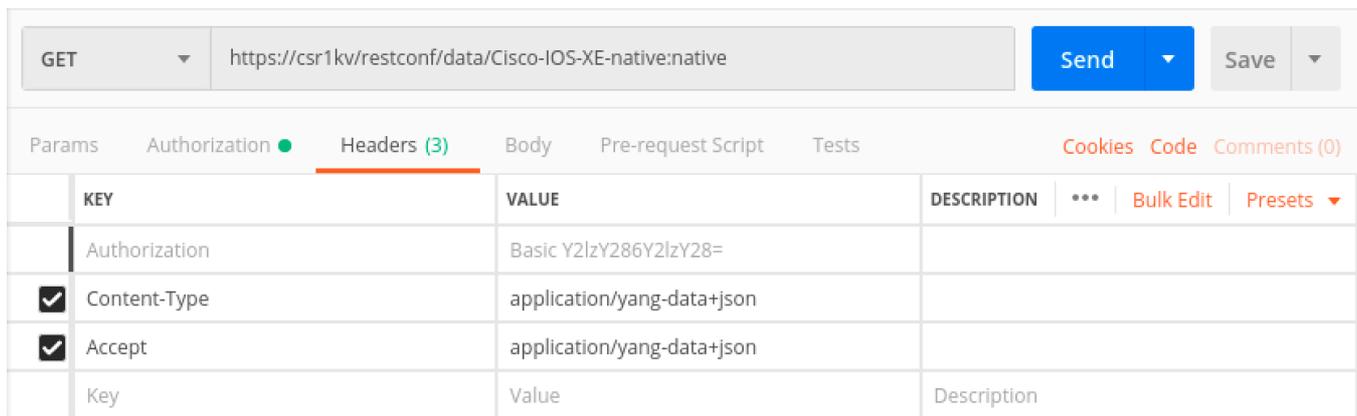
## Step 2

Navigate to the **Headers** tab and enter the two headers we mentioned earlier and set them to their proper values:

*Accept: application/yang-data+json*

*Content-Type: application/yang-data+json*

These two headers are specific to RESTCONF, but it basically means we want to use **JSON**. You can also change **JSON** to **XML** and see how the data differs.



KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
Authorization	Basic Y2lzY286Y2lzY28=				
<input checked="" type="checkbox"/>	Content-Type	application/yang-data+json			
<input checked="" type="checkbox"/>	Accept	application/yang-data+json			
Key	Value	Description			

### Step 3

At this point, we're ready to click **Send**.

Once you click **Send**, you'll see the following response:

The screenshot shows a REST client interface with the following details:

- Request Method: GET
- Request URL: https://csr1kv/restconf/data/Cisco-IOS-XE-native:native
- Environment: No Environment
- Headers (3):
  - Authorization: Basic Y2lzY286Y2lzY28=
  - Content-Type: application/yang-data+json
  - Accept: application/yang-data+json
- Status: 200 OK
- Time: 2514 ms
- Size: 7.78 KB
- Response Body (JSON):

```
1 {
2   "Cisco-IOS-XE-native:native": {
3     "version": "16.11",
4     "boot-start-marker": [
5       null
6     ],
7     "boot-end-marker": [
8       null
9     ],
10    "memory": {
11      "free": {
12        "low-watermark": {
13          "processor": 80526
14        }
15      }
16    },
17    "call-home": {
18      "Cisco-IOS-XE-call-home:contact-email-addr": "sch-smart-licensing@cisco.com",
19      "Cisco-IOS-XE-call-home-profile": {
```

Take several minutes to explore the data that was returned. The data returned is a **JSON** structured response and you can see how the keys and values do indeed map back to more familiar **CLI** commands.

## Step 4

By default, the whole data set is returned with the previous API call. However, RESTCONF supports a query string “*?depth=*” that you can add to **GET** API calls to limit the depth of the set of data returned. Re-run the previous API but append “*?depth=1*” to the URL.

Enter in the following URL: <https://csr1kv/restconf/data/Cisco-IOS-XE-native:native?depth=1>

Click the **Send** Button and execute the API call. Below is a portion of what you’ll see:

The screenshot shows a REST client interface with the following details:

- Method: GET
- URL: <https://csr1kv/restconf/data/Cisco-IOS-XE-native:native?depth=1>
- Status: 200 OK
- Time: 405 ms
- Size: 1007 B

The response body is displayed in JSON format:

```
1 {
2   "Cisco-IOS-XE-native:native": {
3     "version": "16.11",
4     "boot-start-marker": [
5       null
6     ],
7     "boot-end-marker": [
8       null
9     ],
10    "memory": {},
11    "call-home": {},
12    "service": {},
13    "platform": {},
14    "hostname": "csr1kv",
15    "enable": {},
16    "username": [],
17    "ip": {},
18    "cdp": {},
19    "interface": {}
20  }
```

Did you notice how much the data was returned was limited? Feel free to issue both of these API calls again to compare and contrast them.

## Step 5

Make two more API calls to retrieve the version of software and hostname of the device.

Because the hierarchy is very well defined (from the original YANG model), it makes the API very flexible. You simply add the key/values you want to the URL on **GET** requests.

- **Retrieve the OS Version**

URL: `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/version`

- **Retrieve the Device Hostname**

URL: `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/hostname`

Both version and hostname are “root” keys in the response from the previous API calls. See how the URLs map to tree hierarchy of data.

Go ahead and attempt to get those values using Postman.

## Step 6

Let’s look at another example.

The picture below shows the license, which is also a “root” key. Issue an API call to return just the “udi” information.

```
1 {
2   "Cisco-IOS-XE-native:udi": {
3     "pid": "CSR1000V",
4     "sn": "91HPTU0VG4K"
5   }
6 }
```

URL: `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/license/udi`

Notice how much smaller the return data is and that it’s inversely proportional to the size of the URL, all based on the key-value pairs you’re search for and entering in the URL.

## Step 7

Issue an API call to see all configured interfaces on the system.

Method: *GET*

URL: *https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface*

You can see that a list objects (array) is returned that uses right angle square brackets. When this happens, you use the name key's value in the URL to drill down to that given element in the list. For example, the next Step drills down into **GigabitEthernet2**.

## Step 8

Remove **ANY** secondary addresses on **GigabitEthernet2**.

SSH into the router using the Linux terminal command:

```
ssh cisco@csr1kv
```

Configure the router:

```
csr1kv(config)#int gi2
csr1kv(config-if)#no ip address 10.5.1.9 255.255.255.0 secondary
csr1kv(config-if)#no ip address 10.55.1.9 255.255.255.0 secondary
csr1kv(config-if)#end
csr1kv#
```

---

*YOU MUST TYPE "end" to leave configuration mode!!! If you do not you will not see the changes in Postman.*

---

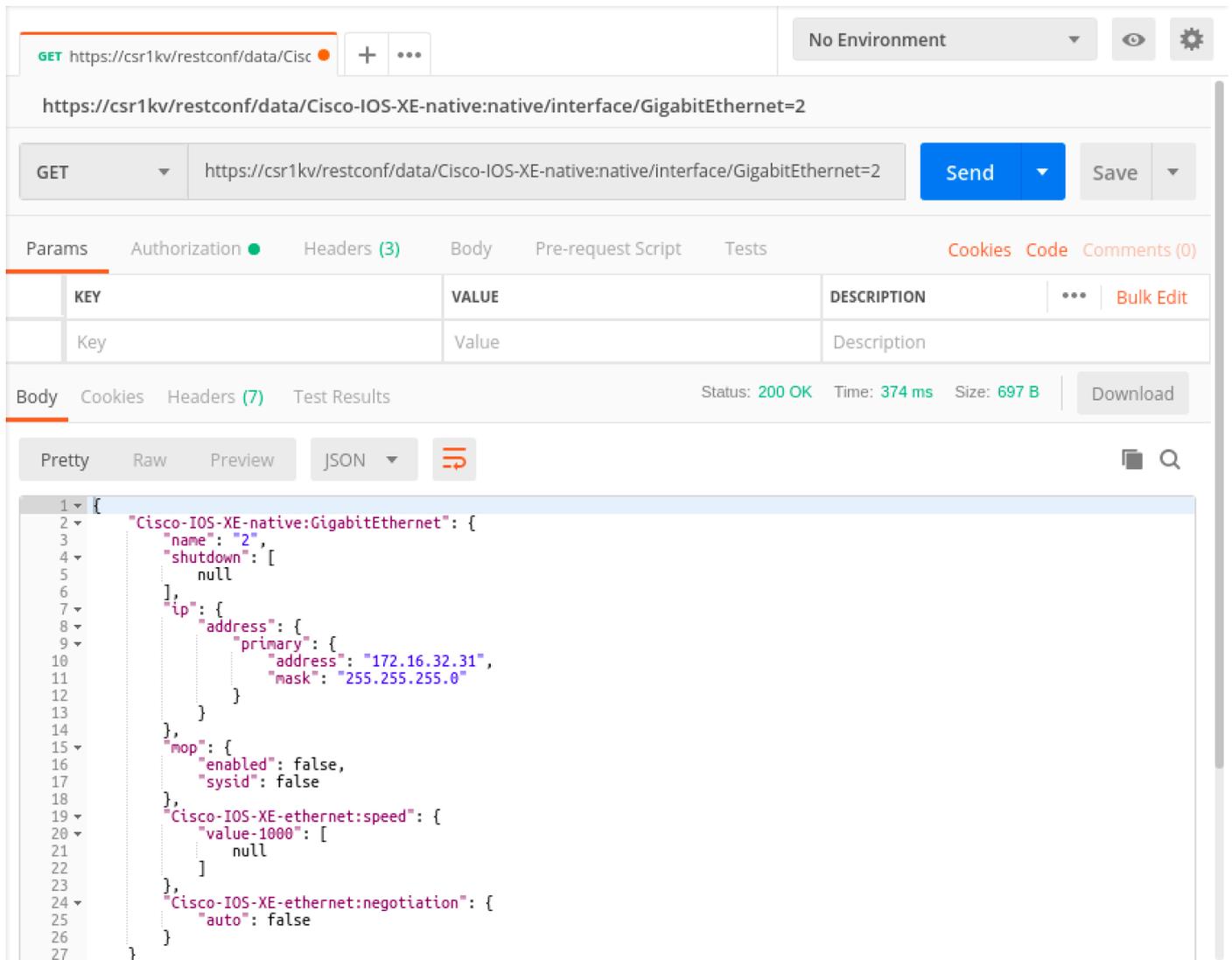
## Step 9

Update the URL to just query **GigabitEthernet2**

URL: `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=2`

Click **Send**

You'll see the following response:



The screenshot shows a REST client interface with the following details:

- Request Method: GET
- Request URL: `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=2`
- Status: 200 OK
- Time: 374 ms
- Size: 697 B

The response body is displayed in JSON format:

```
1 {
2   "Cisco-IOS-XE-native:GigabitEthernet": {
3     "name": "2",
4     "shutdown": [
5       null
6     ],
7     "ip": {
8       "address": {
9         "primary": {
10          "address": "172.16.32.31",
11          "mask": "255.255.255.0"
12        }
13      }
14    },
15    "nop": {
16      "enabled": false,
17      "sysid": false
18    },
19    "Cisco-IOS-XE-ethernet:speed": {
20      "value-1000": [
21        null
22      ]
23    },
24    "Cisco-IOS-XE-ethernet:negotiation": {
25      "auto": false
26    }
27  }
```

Copy and paste the **JSON** response to your clipboard. Simply select it, right click, and click copy. We are going to use this data in the next task.

## Task 3: Making a Configuration Change

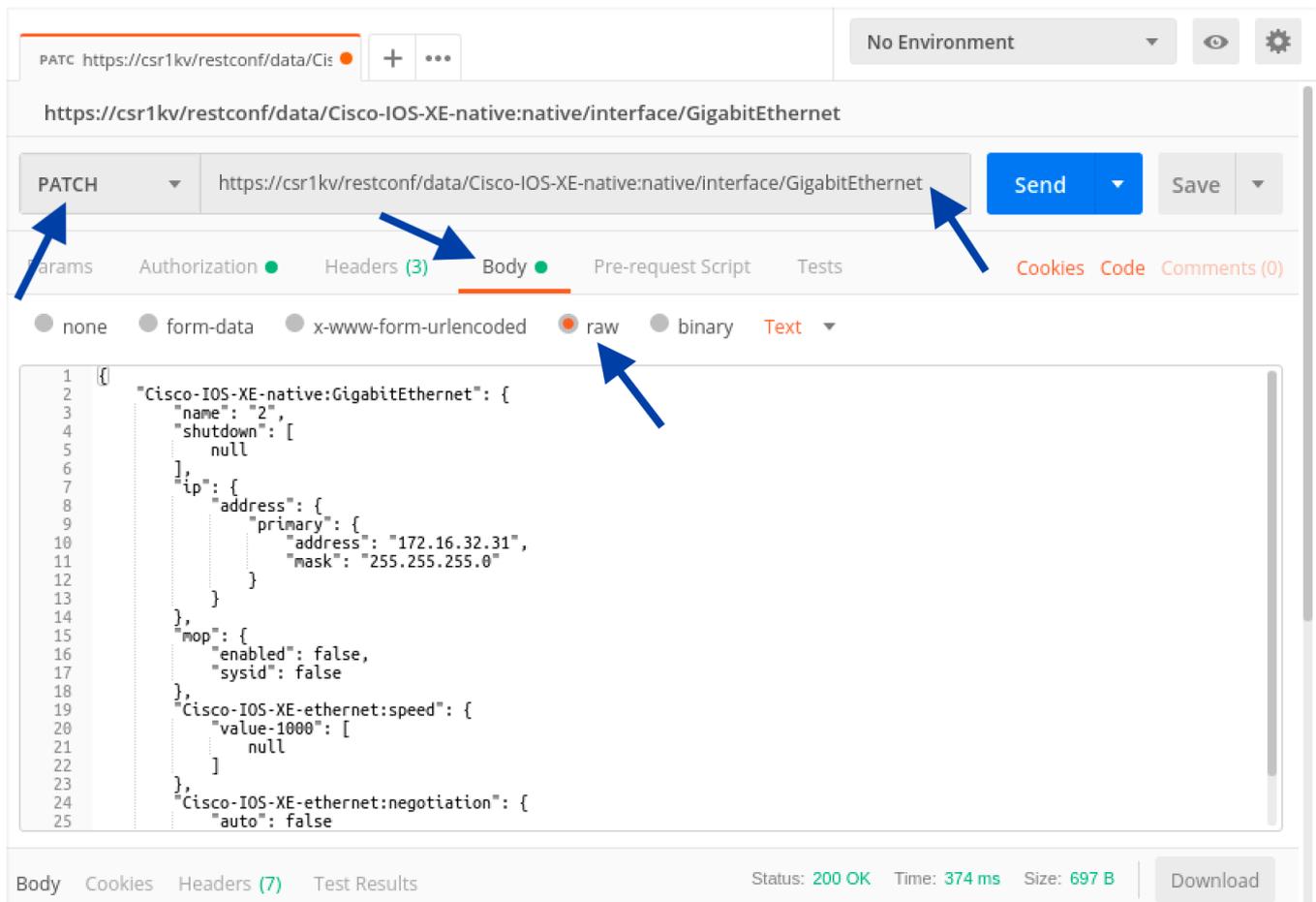
### Step 1

Change the HTTP method to **PATCH**.

URL: `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet`

Copy and paste the **JSON** response in the **Body** tab (next to **Headers**) as shown in the picture below, select **raw** as the **Body Type**:

Remove the “=2” from the end of the URL as shown here and, in the picture, below:



The screenshot shows a REST client interface with the following details:

- Method:** PATCH
- URL:** `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet`
- Body Type:** raw (selected)
- Body Content:**

```
1 {
2   "Cisco-IOS-XE-native:GigabitEthernet": {
3     "name": "2",
4     "shutdown": [
5       null
6     ],
7     "ip": {
8       "address": {
9         "primary": {
10          "address": "172.16.32.31",
11          "mask": "255.255.255.0"
12        }
13      }
14    },
15    "mop": {
16      "enabled": false,
17      "sysid": false
18    },
19    "Cisco-IOS-XE-ethernet:speed": {
20      "value-1000": [
21        null
22      ]
23    },
24    "Cisco-IOS-XE-ethernet:negotiation": {
25      "auto": false
26    }
27  }
28 }
```
- Status:** 200 OK
- Time:** 374 ms
- Size:** 697 B

Click **Send**

At this point you're pushing the same exact configuration the device already has to ensure we have the format of our **JSON** Body correct.

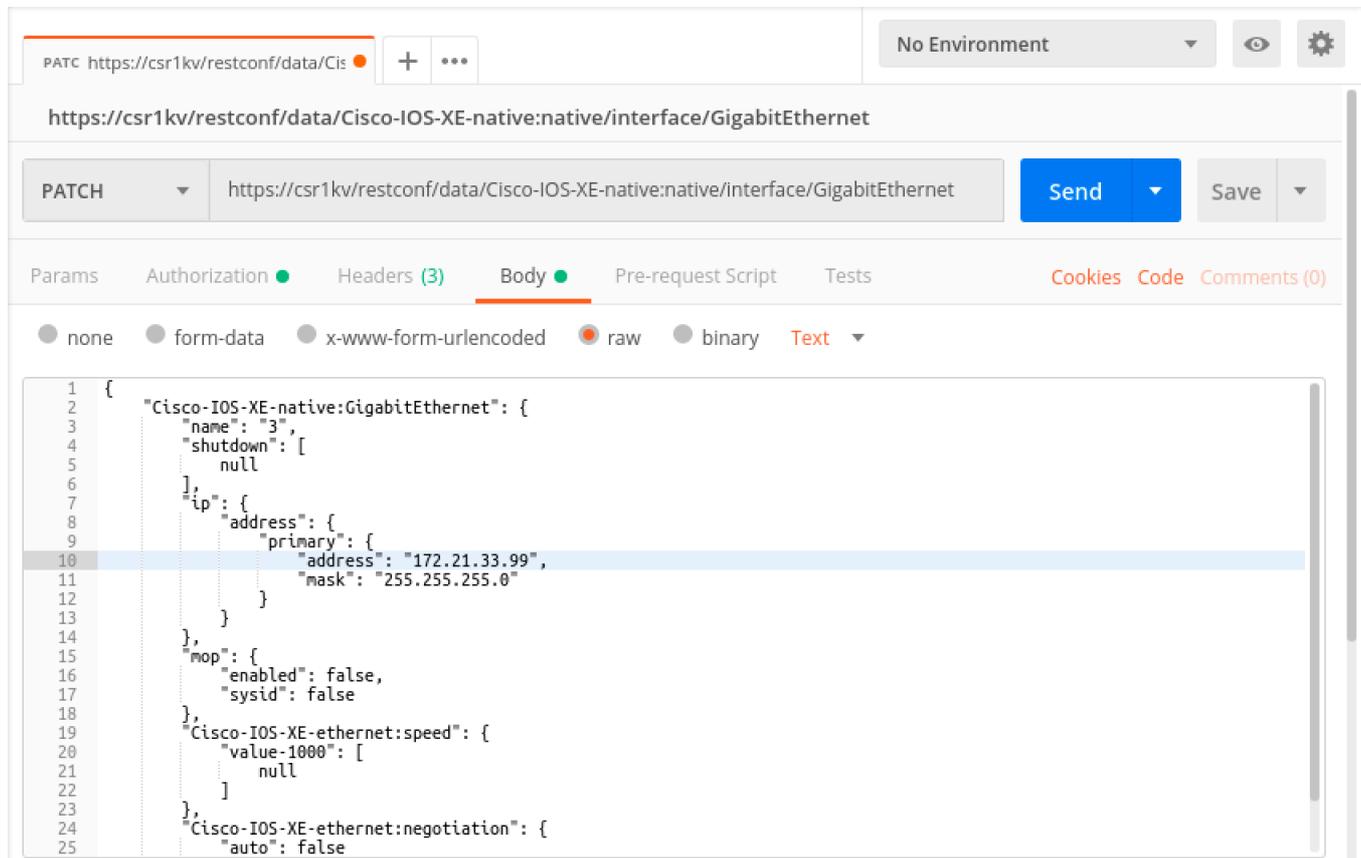
You should have received a **HTTP 200** level message saying this was successful. More specifically you will receive a **HTTP 204** meaning no content returned and request successfully processed.

## Step 2

### Updating the configuration for **GigabitEthernet3**

Change the **“name”** value from **“2”** to **“3”** for the name key in the JSON Body.

Update the IP address and mask to: **172.21.33.99** and **255.255.255.0**



The screenshot shows a REST client interface with the following details:

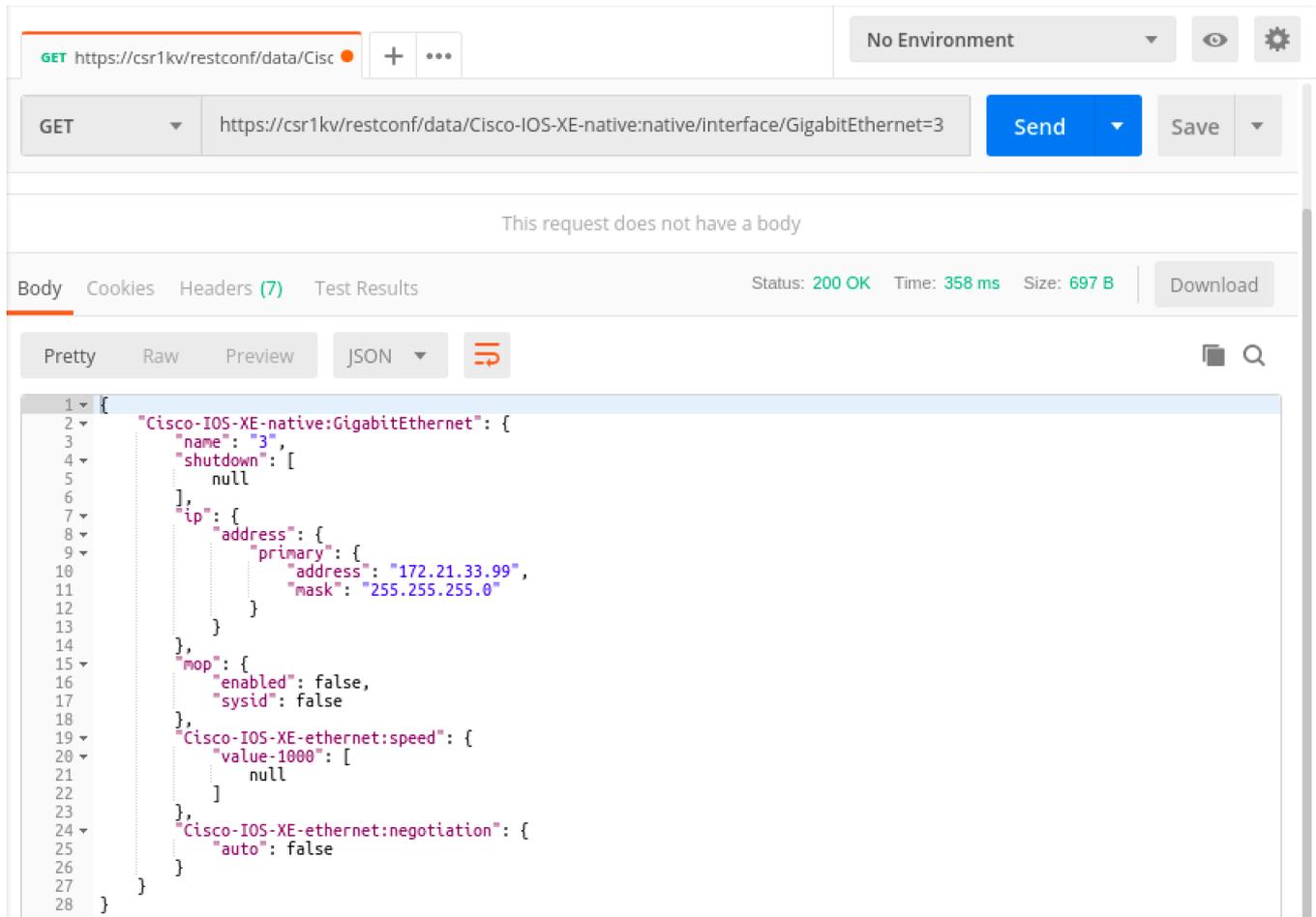
- Method: PATCH
- URL: `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet`
- Body (raw):

```
1 {
2   "Cisco-IOS-XE-native:GigabitEthernet": {
3     "name": "3",
4     "shutdown": [
5       null
6     ],
7     "ip": {
8       "address": {
9         "primary": {
10          "address": "172.21.33.99",
11          "mask": "255.255.255.0"
12        }
13      }
14    },
15    "mop": {
16      "enabled": false,
17      "sysid": false
18    },
19    "Cisco-IOS-XE-ethernet:speed": {
20      "value-1000": [
21        null
22      ]
23    },
24    "Cisco-IOS-XE-ethernet:negotiation": {
25      "auto": false
26    }
27  }
28 }
```

Click **Send**

### Step 3

Verify the changes on **GigabitEthernet3** using Postman using a **GET** request.



The screenshot shows the Postman interface for a RESTCONF GET request. The URL is `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=3`. The response status is 200 OK, with a time of 358 ms and a size of 697 B. The response body is displayed in JSON format, showing the configuration for GigabitEthernet3.

```
1 {
2   "Cisco-IOS-XE-native:GigabitEthernet": {
3     "name": "3",
4     "shutdown": [
5       null
6     ],
7     "ip": {
8       "address": {
9         "primary": {
10          "address": "172.21.33.99",
11          "mask": "255.255.255.0"
12        }
13      }
14    },
15    "mop": {
16      "enabled": false,
17      "sysid": false
18    },
19    "Cisco-IOS-XE-ethernet:speed": {
20      "value-1000": [
21        null
22      ]
23    },
24    "Cisco-IOS-XE-ethernet:negotiation": {
25      "auto": false
26    }
27  }
28 }
```

## Step 4

Manually SSH from the Linux terminal into the router using the following command:

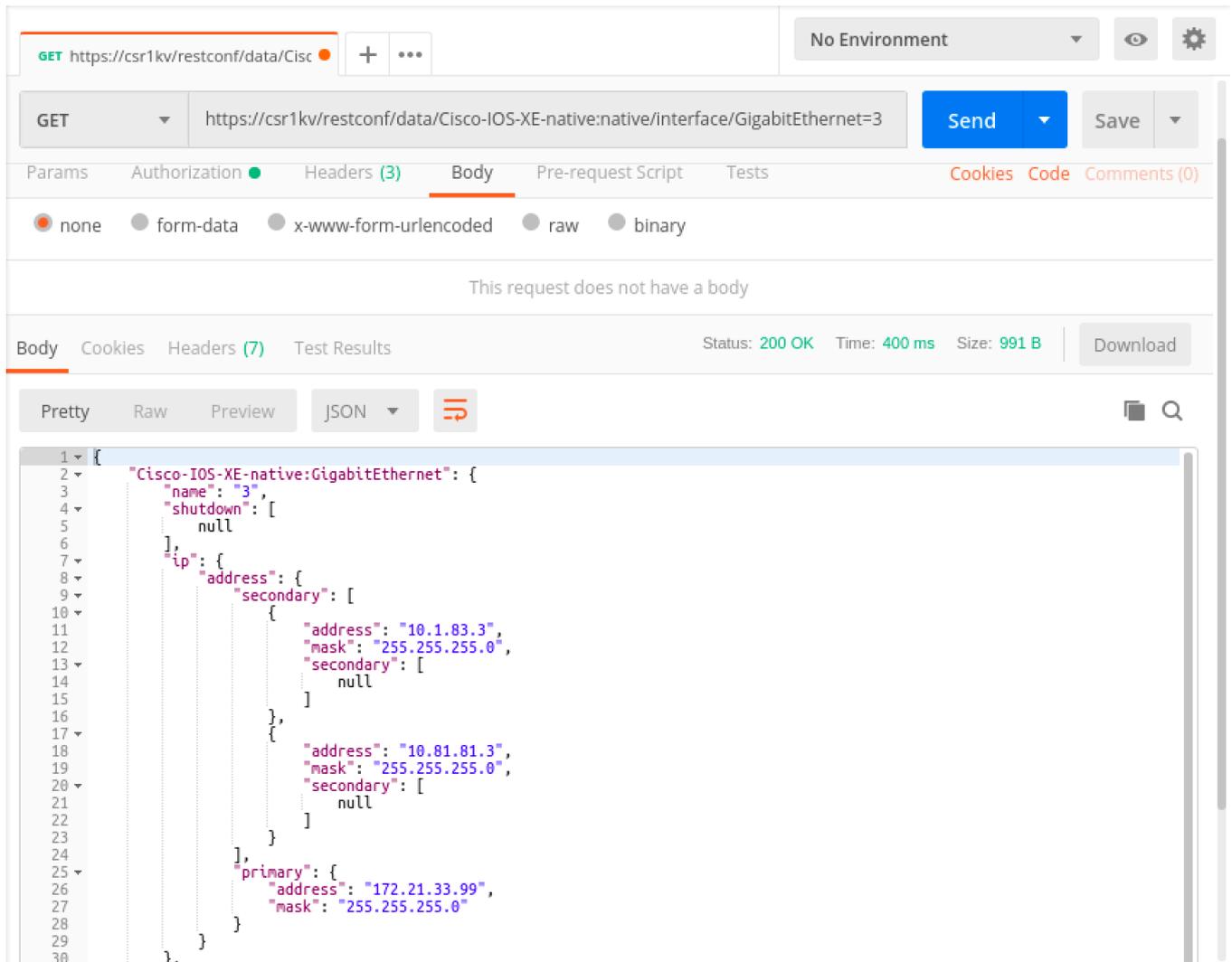
```
ssh cisco@csr1kv
```

Enter the following commands to add two secondary addresses to **GigabitEthernet3**:

```
csr1kv#config t
csr1kv (config)#int GigabitEthernet 3
csr1kv (config-if)#ip address 10.1.81.3 255.255.255.0 secondary
csr1kv (config-if)#ip address 10.81.81.3 255.255.255.0 secondary
csr1kv (config-if)#end
csr1kv#
csr1kv# ! REMEMBER YOU MUST ENTER "END"
```

## Step 5

Verify the changes on **GigabitEthernet3** using Postman using a **GET** request. This should be in your **History** by now so you can easily re-execute it.



The screenshot shows the Postman interface with a GET request to `https://csr1kv/restconf/data/Cisco-IOS-XE-native:interface/GigabitEthernet=3`. The response status is 200 OK, with a time of 400 ms and a size of 991 B. The response body is displayed in JSON format, showing the configuration for GigabitEthernet3. The JSON structure is as follows:

```
1 {
2   "Cisco-IOS-XE-native:GigabitEthernet": {
3     "name": "3",
4     "shutdown": [
5       null
6     ],
7     "ip": {
8       "address": {
9         "secondary": [
10          {
11            "address": "10.1.83.3",
12            "mask": "255.255.255.0",
13            "secondary": [
14              null
15            ]
16          },
17          {
18            "address": "10.81.81.3",
19            "mask": "255.255.255.0",
20            "secondary": [
21              null
22            ]
23          }
24        ],
25        "primary": {
26          "address": "172.21.33.99",
27          "mask": "255.255.255.0"
28        }
29      }
30    }
31  }
```

Ensure you see both secondary addresses as well.

## Step 6

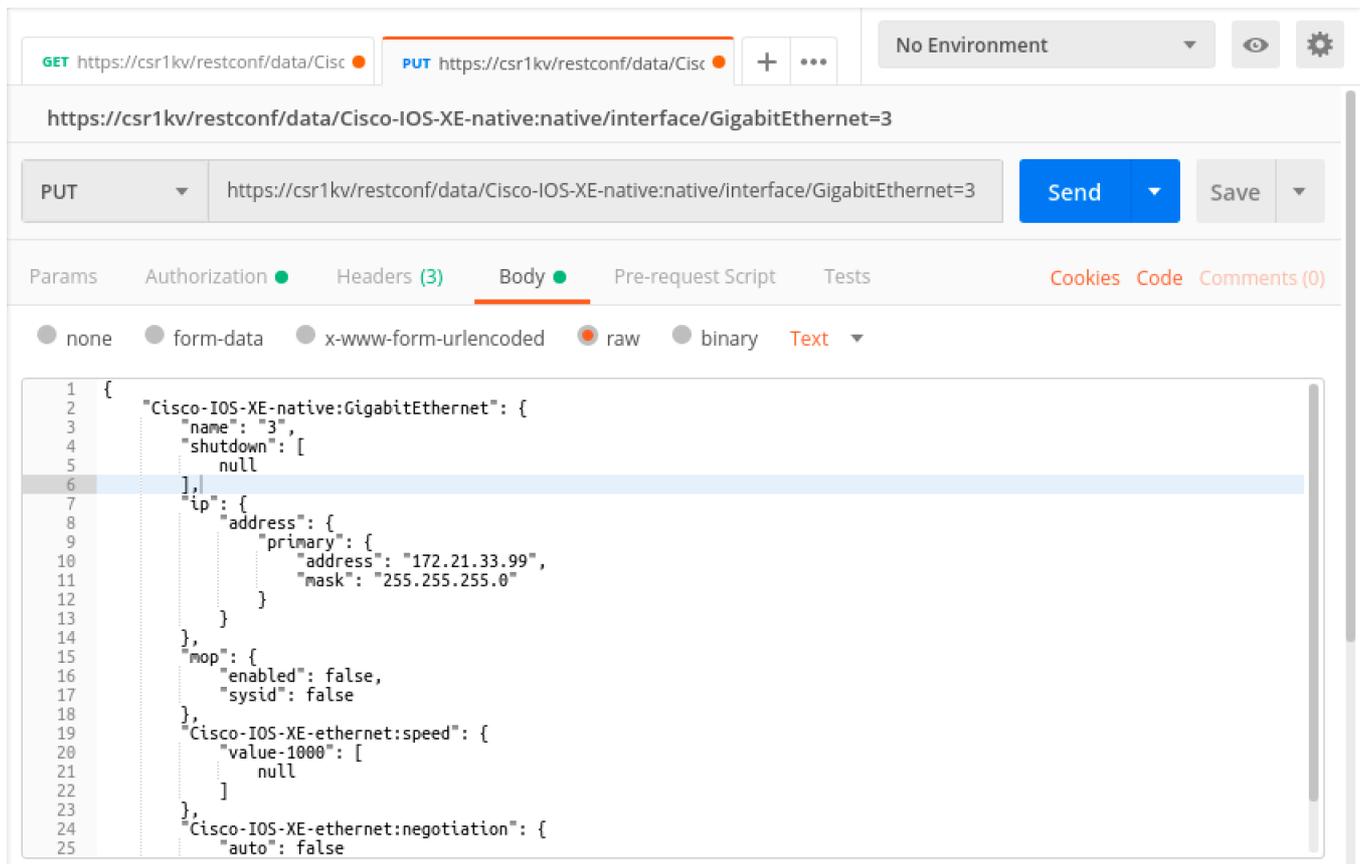
Now we're going to show the power of using **PUT** vs. **PATCH**.

From your **History** on the left, choose the **PATCH** that configured `172.21.33.99/24` on **GigabitEthernet3** and validate the JSON Body does not have any secondary addresses.

Add `=3` back into the end of the URL.

This is required for **PUTs** because we'll be replacing a full element and its children in the configuration hierarchy.

Verify your **URL** and **Body** looks like the following:



The screenshot shows a REST client interface with the following details:

- Method: **PUT**
- URL: `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=3`
- Body Type: **raw** (selected)
- Body Content (JSON):

```
1 {
2   "Cisco-IOS-XE-native:GigabitEthernet": {
3     "name": "3",
4     "shutdown": [
5       null
6     ],
7     "ip": {
8       "address": {
9         "primary": {
10          "address": "172.21.33.99",
11          "mask": "255.255.255.0"
12        }
13      }
14    },
15    "mop": {
16      "enabled": false,
17      "sysid": false
18    },
19    "Cisco-IOS-XE-ethernet:speed": {
20      "value-1000": [
21        null
22      ]
23    },
24    "Cisco-IOS-XE-ethernet:negotiation": {
25      "auto": false
26    }
27  }
28 }
```

Click **Send**.

## Step 7

Perform one more **GET** or manually **SSH** to the router to verify the final configuration on **GigabitEthernet3**. You will have the following configuration:

```
interface GigabitEthernet3
 ip address 172.21.33.99 255.255.255.0
 speed 1000
 no negotiation auto
 no mop sysid
end
```

As you get started using the RESTCONF API, realize there is GREAT POWER using **PUTs** over **PATCHes**. **PUT** literally replaces whatever you are sending in the **JSON** body in the configuration. You are effectively saying *“this is the configuration you want on the device”* without regard for the current configuration which is very much in-line with Cisco’s strategy for intent-based networking.

Take a few more minutes and continue exploring the RESTCONF API on IOS-XE.

# Lab 2: Automating IOS-XE with RESTCONF while using Python requests

In this lab, you will use the Python requests library to automate collecting data as well as configuring IOS-XE using the RESTCONF API.

## Task 1: Getting Familiar with Python Requests

In the previous lab, you learned how to use Postman and make HTTP-Based API calls. You'll now see how that maps directly to using a Python library called `requests` that simplifies working with HTTP-APIs (both RESTful and non-RESTful HTTP APIs). In this first task, you'll prepare basic Python variables and objects that'll be used through this lab.

### Step 1

Open a Linux terminal and then enter the Python Interactive Shell (often referred to use the Python shell) by typing the Linux statement `python`.

```
cisco@student-workstation:~$ python
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Once you see the three greater than signs (`>>>`), this means you're at the Python shell and you can start typing in Python statements. This is a great way to practice Python or making API calls!

### Step 2

Import the required libraries that we need for this lab. This includes `requests` and a helper function within `requests` called `HTTPBasicAuth`.

```
>>> import requests
>>> from requests.auth import HTTPBasicAuth
>>>
```

At this time, we have the required helper functions to make HTTP API calls. This is what `requests` does. You'll see this even more in the next few minutes.

### Step 3

Create three variables called `AUTH`, `MEDIA_TYPE`, and `HEADERS`. The data assigned to these three variables will be sent in each HTTP request we make from Python to the IOS-XE CSR1000V router.

```
>>> AUTH = HTTPBasicAuth('cisco', 'cisco')
>>> MEDIA_TYPE = 'application/yang-data+json'
>>> HEADERS = { 'Accept': MEDIA_TYPE, 'Content-Type': MEDIA_TYPE }
>>>
```

Here is a description of each of these three variables:

`AUTH` - this is using the helper function called `HTTPBasicAuth` to simplify authenticating to the device using standard level 15 credentials.

`MEDIA_TYPE` - this variable is going to be used to define the media-type each header will use. In other words, it tells the router how we're going to send data to it. Because we're sending JSON data modelled from YANG and using the RESTCONF API, this will *mostly* be set to `"application/vnd.yang.data+json"`.

`HEADERS` - We're defining two HTTP headers that will be sent to the router in each session, `Accept` and `Content-Type`. `Accept` is telling the router how to respond and `Content-Type` is telling the router the format of the data we are sending.

## Step 4

We're going to be sending A LOT of GET requests to the CSR1000V, so let's create a function to help us. Functions are used to minimize the amount of duplicate code, so rather than have print statements and API call per URL, you're going to put these statements into a Python function.

```
>>> def get_request(url):
...     response = requests.get(url, auth=AUTH, headers=HEADERS, verify=False)
...     print("API: ", url)
...     print(response.status_code)
...     if response.status_code in [200, 202, 204]:
...         print("Successful")
...     else:
...         print("Error in API Request")
...     print(response.text)
...     print("=" * 40)
...     print("=" * 40)
...
>>> # MAKE SURE TO HIT ENTER TWICE AFTER TYPING IN THE LAST LINE
>>> # 4 SPACE INDENT IS COMMON, BUT AS LONG AS YOUR CONSISTENT, YOU'LL BE OKAY!
```

---

*Note: for convenience and to reduce typos the previous text can be copied from  
~/Desktop/CopyPaste/Lab2-Task1-Step4.txt*

---

We need to pass one object to the function each time, namely a URL. Once the function `get_request` receives the URL, it'll execute the API call as a **GET** request. You see the line `requests.get(...)` which maps directly to the HTTP verb being used. If you wanted to do a HTTP **POST**, you would do `requests.post(...)`.

There are also two *attributes* of the response we're going to use and print out:

`status_code` - this contains the HTTP response code. Valid responses from IOS-XE are 200, 202, and 204.

`text` - this attribute contains the response from the router as a string

Take notice a conditional statement that checks the status and prints a successful or error message based on the API call.

At this point, we're ready to start making API calls. The foundation is in place-- all we need

to do now is pass URLs to the `get_request` function to start issuing HTTP GET requests.

## Task 2: Making a GET request with Python

In the first API call, you're going to start with the same exact API call you made in Postman that retrieves the device's configuration.

### Step 1

Create a variable called `url` and assign it the value of `"http://csr1kv/restconf/api/running/native"`

```
>>> url = "https://csr1kv/restconf/data/Cisco-IOS-XE-native:native"
>>>
```

### Step 2

Pass `url` to the `get_request` function. Remember, the `get_request` function will automatically print the response.

```
>>> get_request(url)
```

Once entered, you'll see the following full response:

```
>>> get_request(url)
API: https://csr1kv/restconf/data/Cisco-IOS-XE-native:native
200
Successful
<output-omitted>
=====
=====
>>>
```

### Step 3

Remember, you can always add `?depth=1` to the URL to retrieve less information about particular elements in the response.

Create a new variable called `url` with the query parameter and call the `get_request` function again.

```
>>> url = "https://csr1kv/restconf/data/Cisco-IOS-XE-native:native?depth=1"
>>> get_request(url)
API: http://csr1kv/restconf/api/running/native?deep
200
Successful
<output-omitted>
=====
=====
>>>
```

Again, we're suppressing the output as you just saw many of these using Postman. The point is to get familiar with performing the same tasks using Python.

## Task 3: Retrieve Interface Configurations

In this task, you're going to make a series of API calls that retrieve interface configuration data. You'll make API calls that retrieve all interfaces, just one interface, and then continue to drill down into very specific data about a given interface.

Issue API calls to the following URLs using the `get_request` function (we'll walk through in each step below):

- `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface`
- `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=2`
- `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=2/ip/`
- `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=1/ip/address/`
- `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=1/ip/address/primary`
- `https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=1/ip/address/primary/address`

### Step 1

Retrieve back configurations for all interfaces configured:

```
>>> url = "https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface"
>>> get_request(url)
API: https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface

200
Successful
<output-omitted>
=====
=====
>>>
```

## Step 2

Narrow down the response to just a single interface:

```
>>> url = "https://csr1kv/restconf/data/Cisco-IOS-XE-
native:native/interface/GigabitEthernet=2"
>>> get_request(url)
API: https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=2
200
Successful
<output-omitted>
=====
=====
>>>
```

## Step 3

Retrieve just the Layer 3 IP configuration for an interface:

```
>>> url = "https://csr1kv/restconf/data/Cisco-IOS-XE-
native:native/interface/GigabitEthernet=2/ip/"
>>> get_request(url)
API: https://csr1kv/restconf/data/Cisco-IOS-XE-
native:native/interface/GigabitEthernet=2/ip/
200
Successful
<output-omitted>
=====
=====
>>>
```

## Step 4

Narrow the request down even more to just receive the IPv4 address and mask of the interface including any secondary interfaces configured on the interface:

```
>>> url = "https://csr1kv/restconf/data/Cisco-IOS-XE-
native:native/interface/GigabitEthernet=1/ip/address/"
>>> get_request(url)
API: https://csr1kv/restconf/data/Cisco-IOS-XE-
native:native/interface/GigabitEthernet=1/ip/address/
200
Successful
<output-omitted>
=====
=====
>>>
```

## Step 5

You can continue to narrow down, or *filter*, the response.

This time, just retrieve the primary IPv4 address and mask.

```
>>> url = "https://csr1kv/restconf/data/Cisco-IOS-XE-
native:native/interface/GigabitEthernet=1/ip/address/primary"
>>> get_request(url)
API: https://csr1kv/restconf/data/Cisco-IOS-XE-
native:native/interface/GigabitEthernet=1/ip/address/primary
200
Successful
<output-omitted>
=====
=====
>>>
```

## Step 6

Finally, make an API call to return just the primary IPv4 address (excluding the mask):

```
>>> url = "https://csr1kv/restconf/data/Cisco-IOS-XE-
native:native/interface/GigabitEthernet=1/ip/address/primary/address"
>>> get_request(url)
API: https://csr1kv/restconf/data/Cisco-IOS-XE-
native:native/interface/GigabitEthernet=1/ip/address/primary/address
200
Successful
<output-omitted>
=====
=====
>>>
```

You should notice how easy it is to *crawl* the response (tree) from a device after seeing the key-value pairs returned from the device.

This is made possible because RESTCONF (and NETCONF) are both modelled driven APIs on IOS-XE.

## Task 4: Get Dynamic Routing Configuration

In this task, you're going to migrate away from using the Python interactive shell and start to use standalone Python scripts to perform similar tasks. Rather than look at interface configurations, you'll extract configuration data for routing protocols.

### Step 1

Exit the Python Interactive Shell

```
>>> exit()  
cisco@student-workstation:~$
```

### Step 2

Open a text editor of your choice on the Student Workstation. There are several included and pre-installed for use. For example, you can use `Atom` or `Notepadqq` or `nano` or `vim`.

If you're not familiar with any of these, you can use `nano`. It offers fairly intuitive editing functionality.

While still on the Linux shell, enter the statement `nano`:

```
cisco@student-workstation:~$ nano
```

You can use the menu options on the bottom after nano editor opens for editing and performing operations in the file.

### Step 3

Enter the following Python statements into `nano` (or the text editor of your choice).

These statements are no different than the ones you typed when you first entered the Python shell.

```
import requests
from requests.auth import HTTPBasicAuth

# Disable SSL Verification Warning because of Private SSL Certificate
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

AUTH = HTTPBasicAuth('cisco', 'cisco')
MEDIA_TYPE = 'application/yang-data+json'
HEADERS = { 'Accept': MEDIA_TYPE, 'Content-Type': MEDIA_TYPE }

def get_request(url):
    response = requests.get(url, auth=AUTH, headers=HEADERS, verify=False)
    print("API: ", url)
    print(response.status_code)
    if response.status_code in [200, 202, 204]:
        print("Successful")
    else:
        print("Error in API Request")
    print(response.text)
    print("=" * 40)
    print("=" * 40)
```

---

*Note: for convenience and to reduce typos the previous text can be copied from  
~/Desktop/CopyPaste/Lab2-Task4-Step3.txt*

---

### Step 4

Before moving forward, let's ensure you don't have any spacing issues. Save and Exit the file by typing `CONTROL+X` (if using `nano`). You will be prompted to save when you try and exit.

Save the file as `cl-restconf.py`.

## Step 5

Execute the Python script from the Linux shell.

```
cisco@student-workstation:~$ python cl-restconf.py
cisco@student-workstation:~$
```

***If you don't see anything, it means your spacing (and indentation) is good. You have no syntax issues.***

Open the file back up using the following command:

```
cisco@student-workstation:~$ nano cl-restconf.py
```

## Step 6

Enter the following statements in your new script. They will be used to retrieve **OSPF** and **BGP** configuration information.

You should enter one `url` and one function call, save the file, exit your text editor, then execute the script for each API call **ONE API AT A TIME**.

```
url = "https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/router/"
get_request(url)

url = "https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/router/bgp=65512"
get_request(url)

url = "https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/router/bgp=65512/bgp/router-id"
get_request(url)

url = "https://crsk1v/restconf/data/Cisco-IOS-XE-native:native/router/ospf=100/router-id"
get_request(url)
```

If you receive any errors, please read them carefully and remember, you can test them in Postman too, if needed.

# Challenge 1: Update Interface

## GigabitEthernet2

In this challenge, you will update the IP address for **GigabitEthernet2** to **172.16.31.202/24**.

1. Create a new Python Script called `update-interface.py`
2. There is no need to copy over the existing `get_request` function for the example (unless you want to, of course)
3. You do need to copy or enter the following into the new script:

```
import requests
from requests.auth import HTTPBasicAuth

# Disable SSL Verification Warning because of Private SSL Certificate
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

AUTH = HTTPBasicAuth('cisco', 'cisco')
MEDIA_TYPE = 'application/yang-data+json'
HEADERS = { 'Accept': MEDIA_TYPE, 'Content-Type': MEDIA_TYPE }
```

4. Use the URL you used to retrieve the configuration for **GigabitEthernet2**
5. Since you're making a change, you **NEED** to pass data in the HTTP Body
6. A great way to see what data needs to be sent is to issue a GET request first to see how you want to structure the data
7. Remember, that it's always `requests.<verb>` - use the verb that is merges or *appends* configuration, but does not REPLACE/CREATE configuration.
8. You will need a statement like this:

```
response = requests.<verb>(url, headers=HEADERS, auth=AUTH, data=payload, verify=False)
```

The `data` key is used when you are making a configuration change. `payload` is a variable that you must define. In your case, you should make `payload` a multi-line string that denotes the object you want to send to the device (just like you did in Postman).

Note: to create a multi-line string in Python, you use triple quotes like this:

```
payload = """
    Long payload
    Large object goes here...
    More data.
    """
```

STOP - ONLY CONTINUE TO THE NEXT PAGE FOR THE SOLUTION

## Challenge 1: Solution

The real challenge here is coming up with the `payload` object. Again, this is easily seen by first querying **GigabitEthernet2**, understanding the format of the data returned, and the pushing the same data back to the device (only the key/value pairs you want to configure).

If you copied over a key called `name`, you should have seen a descriptive error message basically saying the `2` in **GigabitEthernet=2** in the URL and the `name` key could only exist once, thus, we removed it from the **JSON** body. You could have also removed it from the URL as you did with Postman.

### Solution Script:

```
import requests
from requests.auth import HTTPBasicAuth

# Disable SSL Verification Warning because of Private SSL Certificate
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

AUTH = HTTPBasicAuth('cisco', 'cisco')
MEDIA_TYPE = 'application/yang-data+json'
HEADERS = { 'Accept': MEDIA_TYPE, 'Content-Type': MEDIA_TYPE }

url = "https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=2"
payload = """
{
  "Cisco-IOS-XE-native:GigabitEthernet": {
    "ip": {
      "address": {
        "primary": {
          "address": "172.16.31.202",
          "mask": "255.255.255.0"
        }
      }
    }
  }
}
"""
print("Challenge 1:")
response = requests.patch(url, headers=HEADERS, auth=AUTH, data=payload, verify=False)
print(response.status_code)
```

Add in a **GET** request if you'd like too.

---

*Note: Solution script located in ~/Desktop/Solutions/challenge\_1.py*

---

## Challenge 2: Create a Loopback Interface

Create a new loopback interface called `Loopback100` that has the following configuration:

**IP Address: 10.8.1.6**

**Mask: 255.255.255.255**

---

*Note: use Postman as needed to explore **GET** requests to understand the structure needed to send to the device!*

---

STOP - ONLY CONTINUE TO THE NEXT PAGE FOR THE SOLUTION

## Challenge 2: Solution

Solution Script:

```
import requests
from requests.auth import HTTPBasicAuth

# Disable SSL Verification Warning because of Private SSL Certificate
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

AUTH = HTTPBasicAuth('cisco', 'cisco')
MEDIA_TYPE = 'application/yang-data+json'
HEADERS = { 'Accept': MEDIA_TYPE, 'Content-Type': MEDIA_TYPE }

url = "https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface"

# this also works if you remove "name" key from JSON payload:
# https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback/100

payload = """
    {
      "Cisco-IOS-XE-native:Loopback": {
        "name": "100",
        "ip": {
          "address": {
            "primary": {
              "address": "10.8.1.6",
              "mask": "255.255.255.255"
            }
          }
        }
      }
    }
  """

print("Challenge 2:")
response = requests.post(url, headers=HEADERS, auth=AUTH, data=payload, verify=False)
print(response.status_code)
if response.status_code >= 400:
    print("Remember, you can only POST once (on creation)!")

url = "https://csr1kv/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100"
response = requests.get(url, headers=HEADERS, auth=AUTH, verify=False)
print(response.text)
print("Is there a new Loopback with the IP address 10.8.1.6, because it should be.")
```

---

*Note: Solution script located in ~/Desktop/Solutions/challenge\_2.py*

---

# Lab 3: Automating IOS-XE with NETCONF using Python ncclient

In this lab, you will use a Python library called ncclient (NETCONF Client) to automate collecting data as well as configuring IOS-XE using the NETCONF API.

## Task 1: Getting Familiar with Python ncclient

In the previous labs, you learned how to use Postman and Python requests to make HTTP-Based API calls using JSON payloads. You'll now see how that maps to using NETCONF while getting more familiar with how JSON data converts well to XML data of the same data models on IOS-XE.

---

*Note: you could have also used XML encoding in Postman and requests too by changing values of HTTP headers.*

---

In this first task, you'll prepare the framework needed to run scripts using the Python ncclient library - this task is similar to the first one you did with requests too.

### Step 1

Create a script called `cl-netconf.py` and open it in a text editor such as **nano** just like you did in the last lab.

### Step 2

At the top of the file, import the required Python objects that are required to work with ncclient and XML more generally while in Python:

```
from lxml import etree
from ncclient import manager
```

We are going to use the `etree` function within `lxml` to help us parse **XML** and pretty print the response from the Cisco IOS-XE router.

For `ncclient`, we are specifically using the `manager` object that handles all of the session connects and disconnects. Remember, NETCONF is running over SSH so it is a connection-oriented protocol. `manager` handles this connection setup as well as the sending of messages back and forth using proper **NETCONF** operations.

### Step 3

Just below the import statements, create two functions as shown below:

```
def get_request(xmlstring):
    print("XML FILTER:")
    print(xmlstring)
    print("-" * 80)
    with manager.connect(host='csr1kv', port=830,
                        username='cisco', password='cisco',
                        hostkey_verify=False, device_params={},
                        allow_agent=False, look_for_keys=False) as device:

        netconf_get_reply = device.get(('subtree', xmlstring))

    print("NETCONF RESPONSE:" )
    print(etree.tostring(netconf_get_reply.data_ele, pretty_print=True).decode('utf-8'))
    print("=" * 80)
    print("=" * 80)
    print("=" * 80)

def edit_request(xmlstring):
    print("XML CONFIG STRING:" )
    print(xmlstring)
    print("-" * 80)

    with manager.connect(host='csr1kv', port=830,
                        username='cisco', password='cisco',
                        hostkey_verify=False, device_params={},
                        allow_agent=False, look_for_keys=False) as device:

        nc_reply = device.edit_config(target='running', config=xmlstring)

    print("NETCONF RESPONSE:" )
    print(nc_reply)
    print("=" * 80)
    print("=" * 80)
    print("=" * 80)
```

---

*Note: for convenience and to reduce typos the previous text can be copied from  
~/Desktop/CopyPaste/Lab3-Task1-Step3.txt*

---

These two functions are going to be responsible for executing the **NETCONF** requests to the Cisco CSR 1000V. The first function called `get_request` is responsible for issuing **NETCONF GET** operations and the second called `edit_request` is responsible for issuing **NETCONF EDIT** operations. **EDIT** operations are used make configuration changes while **GET** operations are used to retrieve data.

There are several print statements in each function, but only a few key statements that map back to the imports we made using `lxml` and `ncclient`.

We're using what's called a context manager with the `with` statement. This streamlines the opening and closing of sessions for us. The variable `device` represents a device object and whenever you are indented under the `with` statement, the connection is active to the network device. As soon as you are un-indented (back at the left most margin) the connection is automatically closed.

Take note of `manager.connect(...)` statement. We're passing in several arguments namely hostname, credentials, and other parameters relevant for **SSH** because **NETCONF** is using **SSH** as transport. **NETCONF** also uses port 830 by default.

The next statement to consider is the following:

```
netconf_get_reply = device.get(('subtree', xmlstring))
```

This is the statement that actually goes to the devices and issues **NETCONF** `<get>` operations. Remember `manager.connect(...)` establishes the connection-- it is then with the `device.get()` statement that an actual request for data takes place.

There are two main types of filters with **NETCONF**; we're going to use the `subtree` filter. This requires an **XML** object or string that acts as a filter to tell the target device, what to respond back with. Within the function, our filter is called `xmlstring`.

The next statement in the `get_request` function is responsible for pretty printing the output as an **XML** string. `data_ele` is an **XML** Python object that is returned from `ncclient`. We simply take that and convert it to a string, then pretty print it and finally decode the binary string to a UTF-8 string for display.

```
print(etree.tostring(netconf_get_reply.data_ele, pretty_print=True).decode('utf-8'))
```

The last major statement to understand is:

```
nc_reply = device.edit_config(target='running', config=xmlstring)
```

This statement sends configuration objects to the network device. You need to pass in two parameters when using the `edit_config` method within the `device` object. The first is `target`. Recall that **NETCONF** supports three data stores such as *running*, *startup*, and *candidate*. We'll be making our changes directly to the running configuration. The second parameter is called `config`, this must be an **XML** string or object that represents the changes you want to make.

That's enough background for now. It's time to get started.

## Task 2: Retrieving the Running Configuration

In this task, we're going to start with the same API call we started with in Postman and when using Python requests. This is the call retrieve the *running* configuration.

### Step 1

Create a variable called `xml_filter` and assign it the value as denoted below. This is equivalent to doing `/restconf/api/running/native` in the URL when working with RESTCONF.

After creating the variable, send it to the `get_request` function. Use print statements to identify the example running as it will help troubleshoot if needed since you will be adding quite a few more examples to this script.

```
xml_filter = """
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
    </native>
    """

print("NETCONF EXAMPLE 1: ")
get_request(xml_filter)
print("*" * 80)
print("\n")
```

### Step 2

Save the script and execute the script.

```
cisco@student-workstation:~$ python cl-netconf.py
<output-omitted>
```

## Task 3 - Retrieving Interfaces

In this task, we'll issue a request to obtain the configurations for all interfaces as well as for a single interface.

### Step 1

Open the script again in the editor.

Create a variable called `xml_filter` and assign it the value as dented below. This is equivalent to doing `/restconf/data/Cisco-IOS-XE-native:native` in the URL when working with RESTCONF.

After creating the variable, send it to the `get_request` function. Use print statements to identify the example running as it'll help troubleshoot if needed since you'll be adding quite a few more examples to this script.

---

*Note: The only thing changing in these **GET** requests is the actual `xml_filter`.*

---

```
xml_filter = """
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
      <interface>
    </interface>
    </native>
  """

print("NETCONF EXAMPLE 2: ")
get_request(xml_filter)
print("*" * 80)
print("\n")
```

### Step 2

Save the script and execute the script.

```
cisco@student-workstation:~$ python cl-netconf.py
```

```
<output-omitted>
```

### Step 3

Add another filter to just retrieve the configuration for **GigabitEthernet2**.

```
xml_filter = """
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
    <interface>
    <GigabitEthernet>
    <name>2</name>
    </GigabitEthernet>
    </interface>
    </native>
    """

print("NETCONF EXAMPLE 3: ")
get_request(xml_filter)
print("*" * 80)
print("\n")
```

### Step 4

Save the script and execute the script.

```
cisco@student-workstation:~$ python c1-netconf.py
```

```
<output-omitted>
```

## Task 4 - Configuring an Interface (Default operation is merge)

### Step 1

Open the script back up and add a new task that will update the configuration for **GigabitEthernet2**.

Remember RESTCONF PATCH API calls *update* the configuration you're sending to the device. That is the default behaviour of NETCONF, but in NETCONF that is called an operation and more specifically the default is `operation="merge"`.

In order to create a configuration string, create a new variable:

```
nc_config = ""
  <config>
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
      <interface>
        <GigabitEthernet>
          <name>2</name>
          <ip>
            <address>
              <primary>
                <address>172.16.31.202</address>
                <mask>255.255.255.0</mask>
              </primary>
              <secondary>
                <address>10.5.1.9</address>
                <mask>255.255.255.0</mask>
              <secondary/>
            </secondary>
            <secondary>
              <address>10.55.1.9</address>
              <mask>255.255.255.0</mask>
            <secondary/>
          </secondary>
        </address>
      </ip>
    </GigabitEthernet>
  </interface>
</native>
</config>
""
```

---

*Note: for convenience and to reduce typos the previous text can be copied from  
~/Desktop/CopyPaste/Lab3-Task4-Step1.txt*

---

This represents what we want to send to the device.

## Step 2

Add a basic print statement to state Example 4 and then call the `edit_request()` function. Please note that the variable name `nc_config` is being used for configuration strings instead of `xml_filter`.

```
print("NETCONF EXAMPLE 4: ")
edit_request(nc_config)
print("*" * 80)
print("\n")
```

## Step 3

Save the script and execute the script.

```
cisco@student-workstation:~$ python cl-netconf.py
<output-omitted>
```

## Step 4

Re-issue a GET request to ensure the configuration was successful.

```
xml_filter = """
  <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
    <interface>
      <GigabitEthernet>
        <name>2</name>
      </GigabitEthernet>
    </interface>
  </native>
"""

print("NETCONF EXAMPLE 5: ")
get_request(xml_filter)
print("*" * 80)
print("\n")
```

## Step 5

Save the script and execute the script.

```
cisco@student-workstation:~$ python cl-netconf.py
<output-omitted>
```

## Task 5 - Configuring an Interface using Replace Operation

### Step 1

Open the script back up and add a new task that will update the configuration for **GigabitEthernet2**. This time we are going to use a **NETCONF** *replace* operation. This is analogous to a **PUT** in **RESTCONF**.

We are going to *replace* the full configuration on the interface automatically removing secondary addresses by only pushing a primary address.

In order to create a configuration string, create a new variable:

```
nc_config = ""
  <config>
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
      <interface>
        <GigabitEthernet>
          <name>2</name>
          <ip operation="replace">
            <address>
              <primary>
                <address>172.16.33.99</address>
                <mask>255.255.255.0</mask>
              </primary>
            </address>
          </ip>
        </GigabitEthernet>
      </interface>
    </native>
  </config>
  ""
```

---

*Note: for convenience and to reduce typos the previous text can be copied from  
~/Desktop/CopyPaste/Lab3-Task5-Step1.txt*

---

Pay special attention to this line:

```
<ip operation="replace">
```

This is how you change the behaviour from the default of `merge` to `replace`.

## Step 2

Make the request to the device:

```
print("NETCONF EXAMPLE 6: ")
edit_request(nc_config)
print("*" * 80)
print("\n")
```

Save and Execute the script.

```
cisco@student-workstation:~$ python cl-netconf.py
<output-omitted>
```

## Step 3

Issue a request to validate the changes:

```
xml_filter = """
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
    <interface>
    <GigabitEthernet>
    <name>2</name>
    </GigabitEthernet>
    </interface>
    </native>
    """

print("NETCONF EXAMPLE 7: ")
get_request(xml_filter)
print("*" * 80)
print("\n")
```

## Step 4

Save the script and execute the script.

```
cisco@student-workstation:~$ python cl-netconf.py
<output-omitted>
```

## Task 6 - Retrieving all dynamic routing configuration

By now, you should be able use Postman, understand the hierarchy in **JSON** and see how that maps to **XML**. Remember, you can also use Postman and change the Headers to use **XML** instead of **JSON** to see how to model for **NETCONF** too.

### Step 1

Create a filter string that queries the device for its "router" configuration. The response will contain the configuration for **BGP** and **OSPF**.

```
xml_filter = """
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
      <router>
      </router>
    </native>
  """

print("NETCONF EXAMPLE 8: ")
get_request(xml_filter)
print("*" * 80)
print("\n")
```

### Step 2

Save the script and execute the script.

```
cisco@student-workstation:~$ python cl-netconf.py

<output-omitted>
```

# Challenge 3: Replace the Complete routing configuration

The current OSPF configuration is:

Process: 100

Router ID: 1.1.1.1

Configured Network: 10.0.0.100 0.0.0.0 in area 0

The current BGP configuration is:

ASN: 65512

Router ID: 1.1.1.2

BGP Log Changes is enabled

Network Advertised: 10.0.0.0

Make one API call to ensure the final routing configuration is the following:

The final OSPF configuration is:

Process: 200

Router ID: 200.1.1.1

Configured Network: 100.0.0.100 0.0.0.0 in area 0

STOP - ONLY CONTINUE TO THE NEXT PAGE FOR THE SOLUTION

## Challenge 3: Solution

Solution Script:

```
from lxml import etree
from ncclient import manager

def get_request(xmlstring):
    print("XML FILTER:")
    print(xmlstring)
    print("-" * 80)
    with manager.connect(host='csr1kv', port=830,
                        username='cisco', password='cisco',
                        hostkey_verify=False, device_params={},
                        allow_agent=False, look_for_keys=False) as device:

        netconf_get_reply = device.get(('subtree', xmlstring))

    print("NETCONF RESPONSE:" )
    print(etree.tostring(netconf_get_reply.data_ele, pretty_print=True).decode('utf-8'))
    print("=" * 80)
    print("=" * 80)
    print("=" * 80)

def edit_request(xmlstring):
    print("XML CONFIG STRING:" )
    print(xmlstring)
    print("-" * 80)

    with manager.connect(host='csr1kv', port=830,
                        username='cisco', password='cisco',
                        hostkey_verify=False, device_params={},
                        allow_agent=False, look_for_keys=False) as device:

        nc_reply = device.edit_config(target='running', config=xmlstring)

    print("NETCONF RESPONSE:" )
    print(nc_reply)
    print("=" * 80)
    print("=" * 80)
    print("=" * 80)

nc_config = """
<config>
  <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
    <router operation="replace">
      <ospf xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ospf">
        <id>200</id>
        <router-id>200.1.1.1</router-id>
        <network >
          <ip>100.0.0.100</ip>
          <mask>0.0.0.0</mask>
          <area>0</area>
        </network>
      </ospf>
    </router>
  </native>
</config>
"""
```

```
        </native>
    </config>
"""

print "ISSUING CHALLENGE REQUEST"
edit_request(nc_config)

xml_filter = """
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
        <router>
        </router>
    </native>
"""

print "ISSUING CHALLENGE VALIDATION"
get_request(xml_filter)
```

Take note of the most important line here:

```
<router operation="replace">
```

If you do not have the operation as replace, you would have simply added another OSPF process to the router!

---

*Note: Solution script located in ~/Desktop/Solutions/challenge\_3.py*

---



**Americas Headquarters**  
Cisco Systems, Inc.  
San Jose, CA

**Asia Pacific Headquarters**  
Cisco Systems (USA) Pte. Ltd.  
Singapore

**Europe Headquarters**  
Cisco Systems International BV Amsterdam,  
The Netherlands

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco Website at [www.cisco.com/go/offices](http://www.cisco.com/go/offices).

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, go to this URL: [www.cisco.com/go/trademarks](http://www.cisco.com/go/trademarks). Third party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1110R)

**Cisco**live!