Network APIs

Agenda

- HTTP-Based APIs
- non-RESTful HTTP-Based APIs
  - Cisco Nexus NX-API
  - Arista eAPI
- RESTful HTTP-Based APIs
  - Cisco IOS-XE RESTCONF
  - Using Postman
- Consuming HTTP-Based APIs with Python requests

HTTP-Based APIs

Network APIs

HTTP-Based APIs

There are two main types of HTTP-Based APIs:

- RESTful HTTP-Based APIs
- non-RESTful HTTP-Based APIs

In other words, those that adhere to the principles of REST and those that do not.

Both use HTTP(s) as transport.

RESTful APIs

- The structure of modern web-based REST APIs came from a PhD paper called Architectural Styles and the Design of Network-based Software Architectures by Roy Fielding in 2000.
- Goal is to define the detail of working with networked systems on the Internet that use the architecture defined as REST
- REST architecture includes six (6) constraints that must be adhered to. Three (3) of them that help understand REST for this course:
  - Client-Server
  - Stateless
  - Uniform interface

REST Architecture

- **Client-Server** - Having a client-server architecture allows for portability and **changeability of client applications** without the server components being changed. This could mean having different clients that consume the server resources (back-end API).
- **Stateless** - the communication between the client and server must be stateless. **Clients** that use stateless forms of communications **must send all data required for the server to understand and perform the requested operation in a single request**. This is in contrast to interfaces such as SSH where there is a persistent connection between a client and a server.
- **Uniform interface** - individual resources in scope within an API call are identified in HTTP request message. For example, in RESTful HTTP-based systems, **the URL used will reference a particular resource**. In the context of networking, the resource maps to a network device construct such as a hostname, interface, routing protocol configuration, or any other *resource* that exists on the device. The uniform interface also states that the client should have enough information about a resource to create, modify, or delete a resource.

HTTP Request Types

RESTful and non-RESTful HTTP APIs use standard HTTP Request Types.

Request Type	Description
GET	Retrieve a specified resource
PUT	Create or replace a resource
PATCH	Create or update a resource object
POST	Create a resource object
DELETE	Delete a specified resource

HTTP Request Types

- In the context of networking, we can think of these request types as the following:
  - GET: obtaining configuration or operational data
  - PUT, POST, PATCH: making a configuration change
  - DELETE: removing a particular configuration

Sample HTTP Requests

- Authentication Type
- HTTP Request Type
- URL
- Headers
  - Accept-Type
  - Content-Type
- Data (Body)

Example 1:

```
Basic Auth: ntc/ntc123
Request: GET
Accept-Type: application/json
URL: http://device/path/to/resource
```

Example 2:

```
Basic Auth: ntc/ntc123
Request: POST
Content-Type: application/json
URL: http://device/path/to/resource
Body: {'interface': "Eth1", "admin_state": "down"}
```

Take note of the body

HTTP Response Codes

Response Code	Description
2XX	Successful
4XX	Client Error
5XX	Server Error

Note: the response code types for HTTP-based APIs are no different than standard HTTP response codes.

Data Encoding

Data is sent over the wire as XML or JSON

- JavaScript Object Notation (JSON)
- Open Standard for data communication
- Uses *name*:*value* pairs
- Maps directly to Python dictionaries

```
{
  "ins_api": {
    "type": "cli_show",
    "version": "1.2",
    "sid": "eoc",
    "outputs": {
      "output": {
        "input": "show hostname",
        "msg": "Success",
        "code": "200",
        "body": {
          "hostname": "nxos-spine1.ntc.com"
        }
      }
    }
  }
}
```

Working with JSON in Python

- Remember `json.dumps(variable, indent=4)`
- We've been using it to pretty print a dictionary
    - It's also serializing the dictionary as a JSON string

```
>>> facts = {'vendor': 'cisco'}
>>>
>>> import json
>>>
>>> test = json.dumps(facts, indent=4)
>>>
>>> type(test)
<type 'str'>
>>>
```

What about the reverse?

- Take a JSON string and consume as a Python dictionary.

Working with JSON in Python

```
#!/usr/bin/env python

import json

if __name__ == "__main__":
    facts = {
        'hostname': 'nxos-spine1',
        'os': '7.3',
        'location': 'New_York'
    }
    print(facts)
    print(json.dumps(facts, indent=4))
    print(facts['os'])
```

```
$ python json_test.py
{'hostname': 'nxos-spine1', 'os': '7.3', 'location':
'New_York'}
{
    "hostname": "nxos-spine1",
    "os": "7.3",
    "location": "New_York"
}
7.3
```

Working with JSON in Python

- JSON strings are not dictionaries
- The **loads** function from **json** converts it into a dictionary

```
facts = '{"hostname": "nxos-spine1", "os": "7.3", "location":
print(facts)
print(type(facts))
print(facts['os'])

factsd = json.loads(facts)
print(factsd)
print(factsd['os'])
```

```
$ python json_test2.py
```

```
{"hostname": "nxos-spine1", "os": "7.3", "location":
"New_York"}
```

```
<type 'str'>
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string indices must be integers, not str
```

```
{"hostname": "nxos-spine1", "os": "7.3", "location":
"New_York"}
```

```
7.3
```

Getting Familiar with JSON Output

- Supported by many vendors who implement web based (REST) APIs
- Certain CLIs allow you to pipe commands to JSON

```
nxos-spine1# show hostname
nxos-spine1.ntc.com
nxos-spine1#
```

```
nxos-spine1# show vlan brief

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------
-------------------
1    default                          active    Eth2/5,
Eth2/6
100  web_vlan                         active

nxos-spine1#
```

```
nxos-spine1# show hostname | json
{
  "hostname": "nxos-spine1.ntc.com"
}
```

```
nxos-spine1# show vlan brief | json
{
  "TABLE_vlanbriefxbrief": {
    "ROW_vlanbriefxbrief": [
      {
        "vlanshowbr-vlanid": 16777216,
        "vlanshowbr-vlanid-utf": 1,
        "vlanshowbr-vlanname": "default",
        "vlanshowbr-vlanstate": "active",
      },
      {
        "vlanshowbr-vlanid": 1677721600,
        "vlanshowbr-vlanid-utf": 100,
        "vlanshowbr-vlanname": "web_vlan",
      }
    ]
output modified for brevity
```

Getting Familiar with JSON Output

Take the output from the CLI and copy into the Python shell for testing

```
>>> test_var = """
... {
...   "hostname": "nxos-spine1.ntc.com"
... }
... """
>>>
```

Perform your tests

```
>>> type(test_var)
<type 'str'>
>>>
>>> response = json.loads(test_var)
>>>
>>> type(response)
<type 'dict'>
>>>
>>> print(response['hostname'])
nxos-spine1.ntc.com
>>>
```

non-RESTful HTTP APIs

Network APIs

What makes APIs non RESTful?

In the context of networking, pay attention to the following:

- The URL is always the same
- The HTTP Request Type (verb) is always the same
  - Are you simply collecting data using a HTTP POST?
  - Are you making a change using a HTTP GET?

non-RESTful HTTP-Based APIs

We are going to look at two non-RESTful APIs: **Cisco Nexus NX-API (CLI)** and **Arista eAPI**.

- All API requests are POSTs
- All API requests use the same URL

Cisco Nexus NX-API

Enable NX-API:

```
feature nxapi
```

Configure ports as needed:

```
nxapi https port 8443
nxapi http port 8080
```

Certain platforms require a command to enable the **sandbox**:

```
nxapi sandbox
```

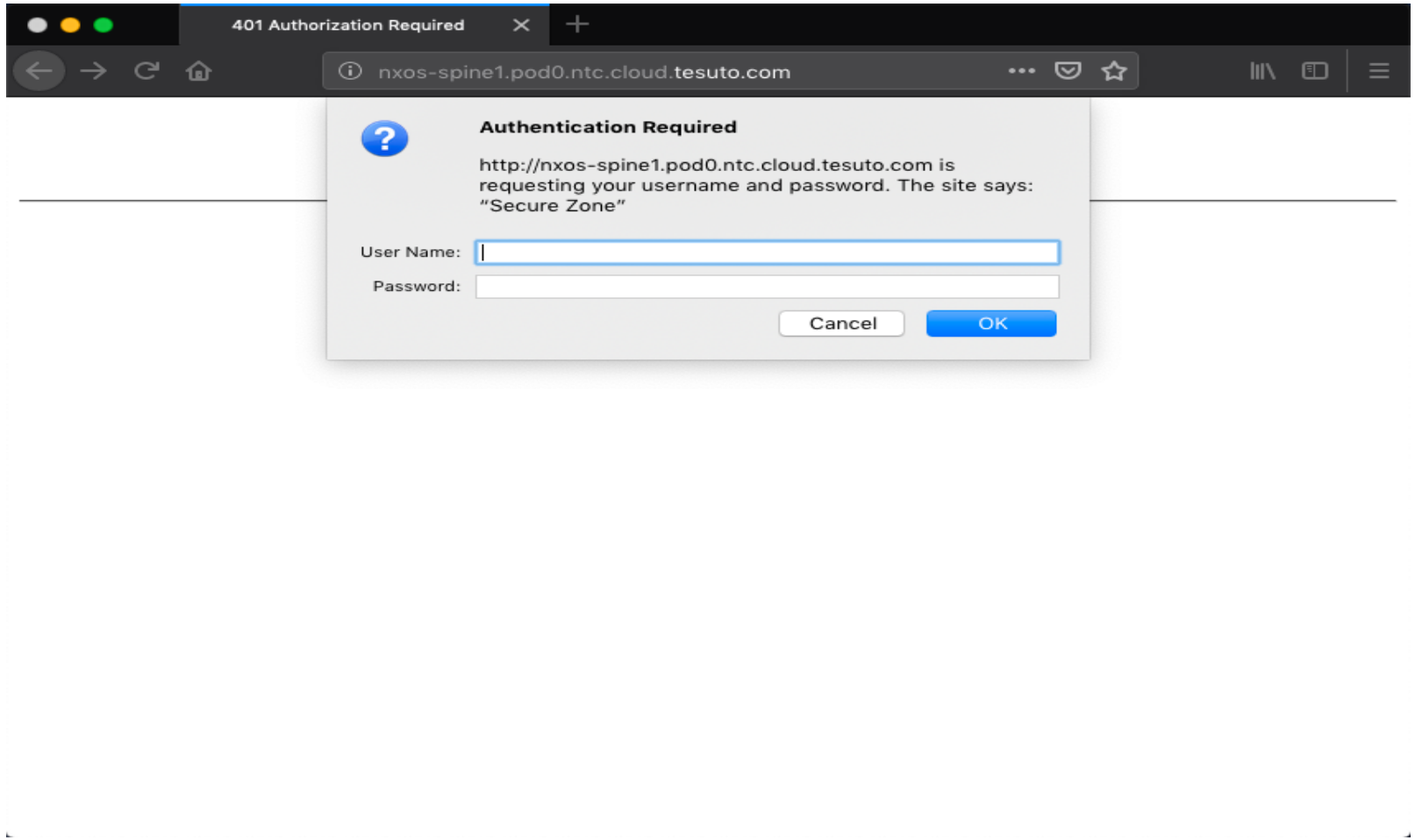
Certain platforms support VRF support:

```
n9k2(config)# nxapi ?
  certificate  Https certificate configuration
  http         Http configuration
  https        Https configuration
  use-vrf      Vrf to be used for nxapi communication
```

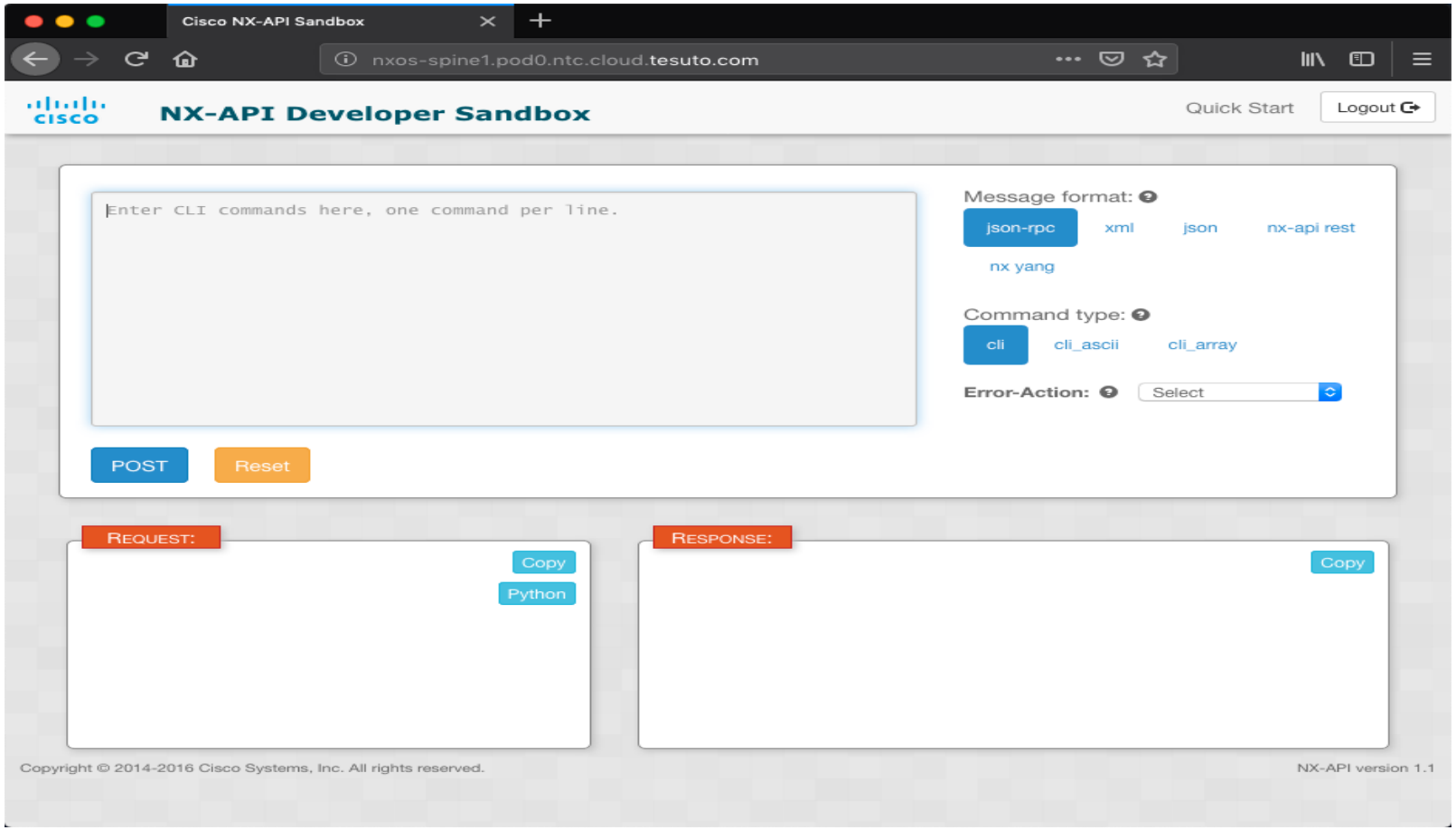
Cisco NX-API Developer Sandbox

- On-box web utility that allows you to practice making API calls
- Visually see response objects before writing code
- Simply browse to the Nexus switch using a web browser

Cisco NX-API Developer Sandbox



Cisco NX-API Developer Sandbox



Arista eAPI

Enable eAPI

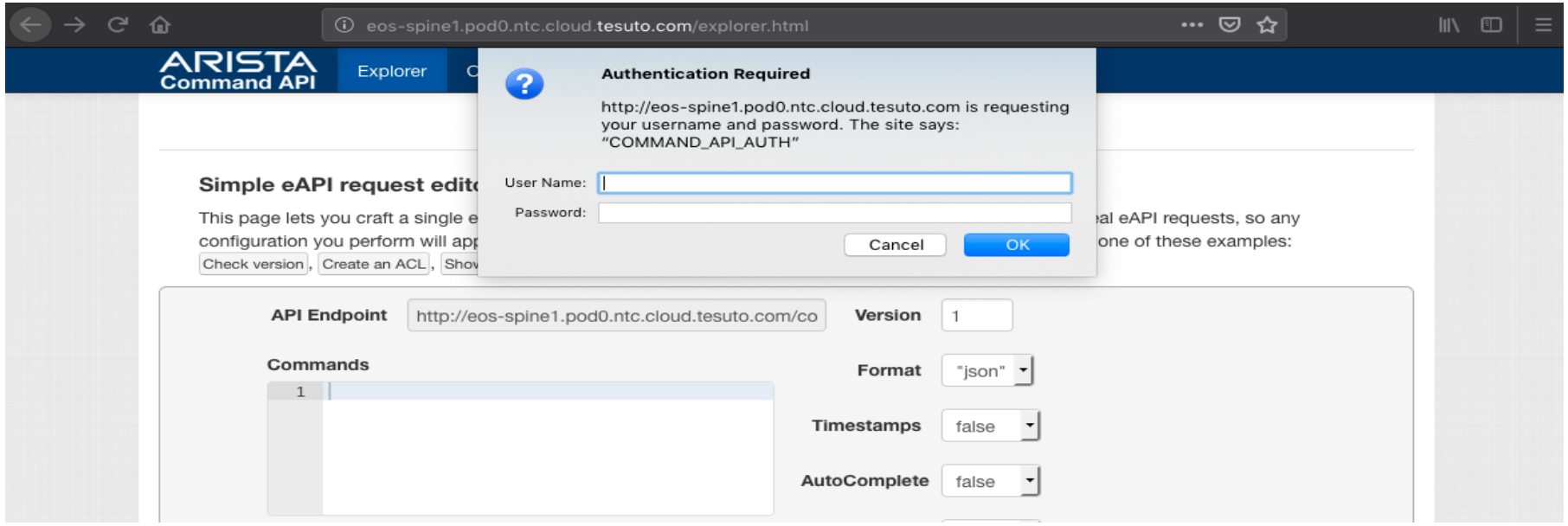
```
management api http-commands
   protocol http
   no shutdown
   vrf MANAGEMENT
      no shutdown
!
```

Arista eAPI Command Explorer

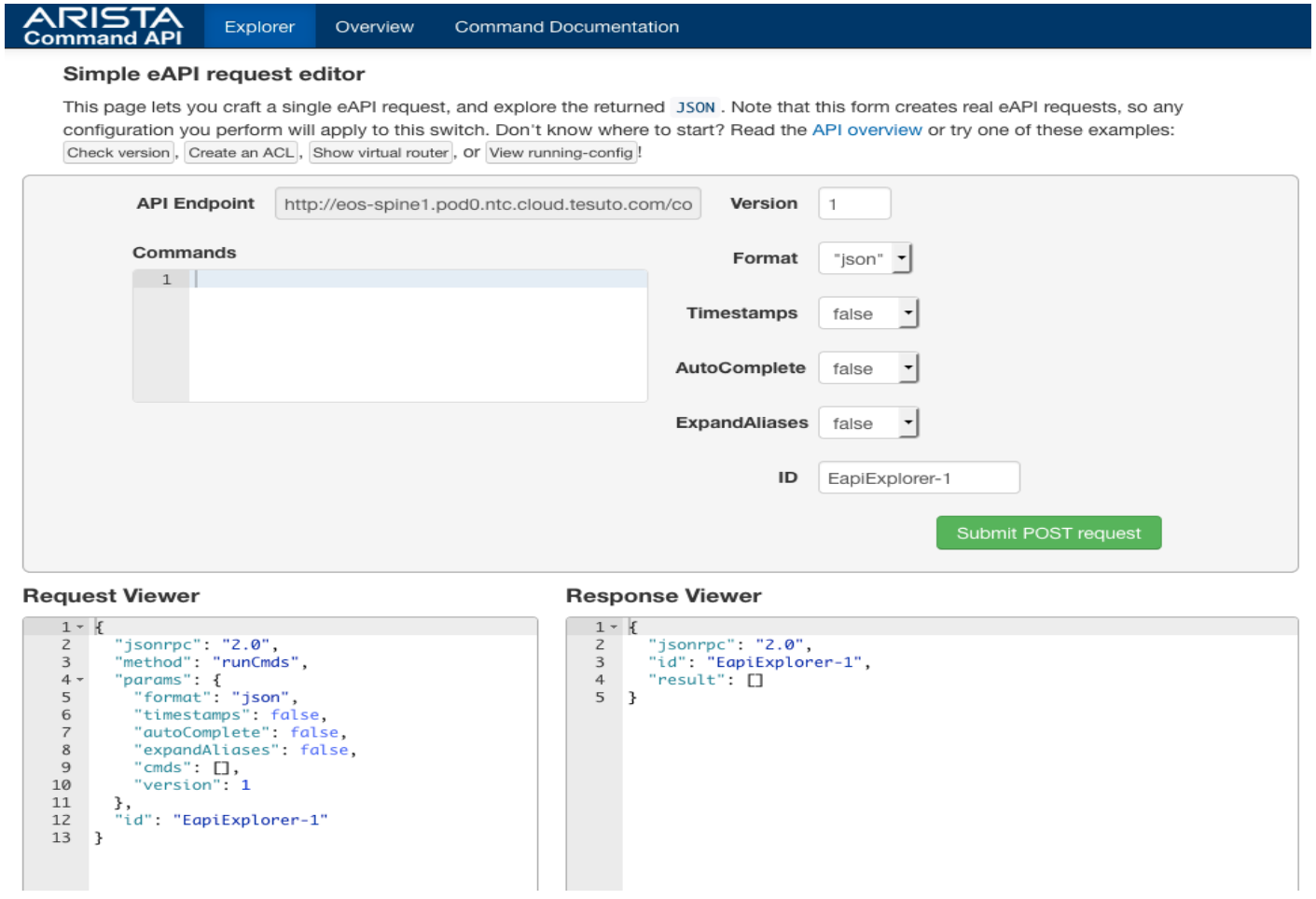
- On-box web utility that allows you to practice making API calls
- Visually see response objects before writing code
- Simply browse to the Arista switch using a web browser

eAPI Command Explorer

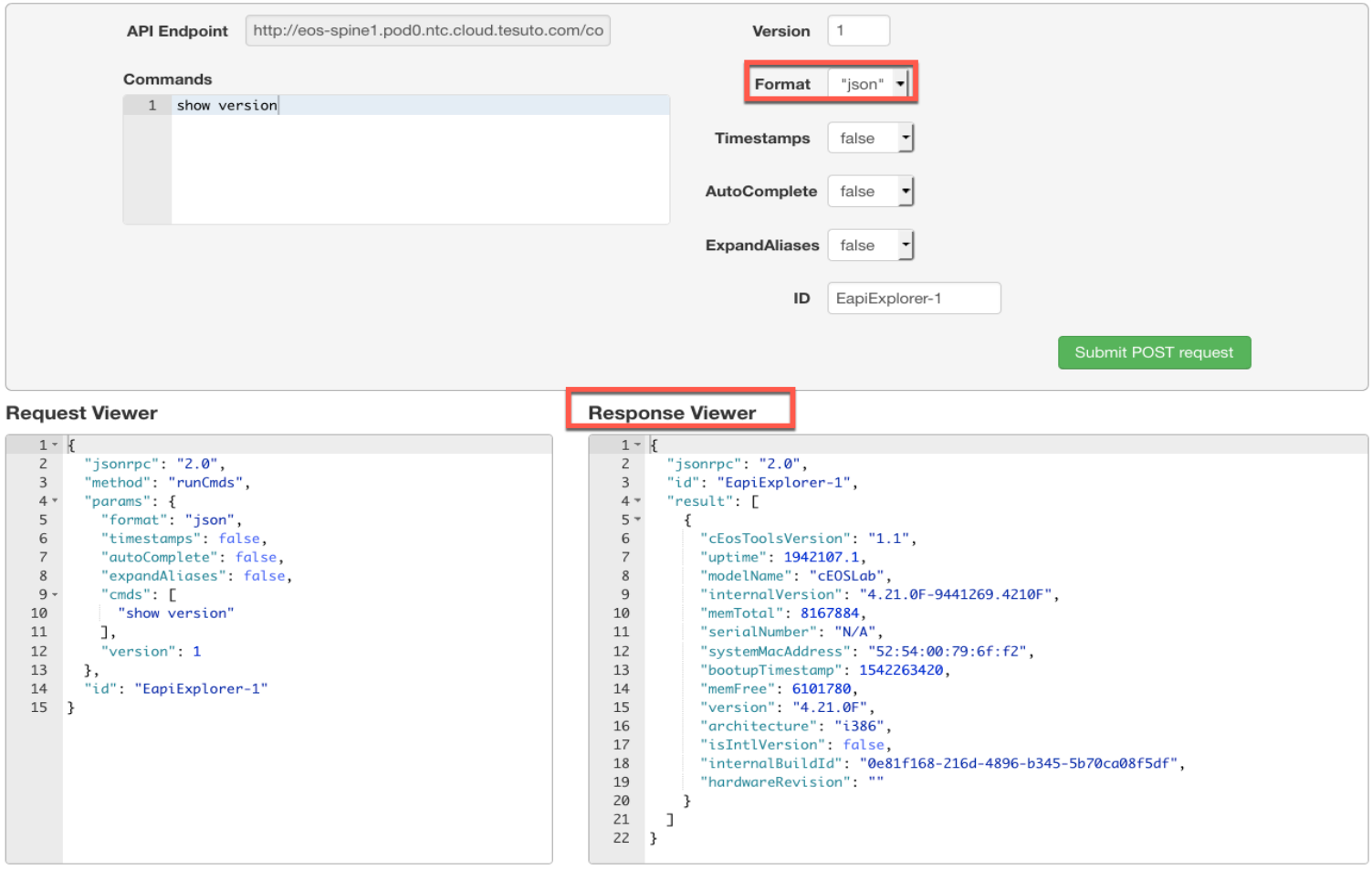
- Browser to http(s)://switch_ip_name



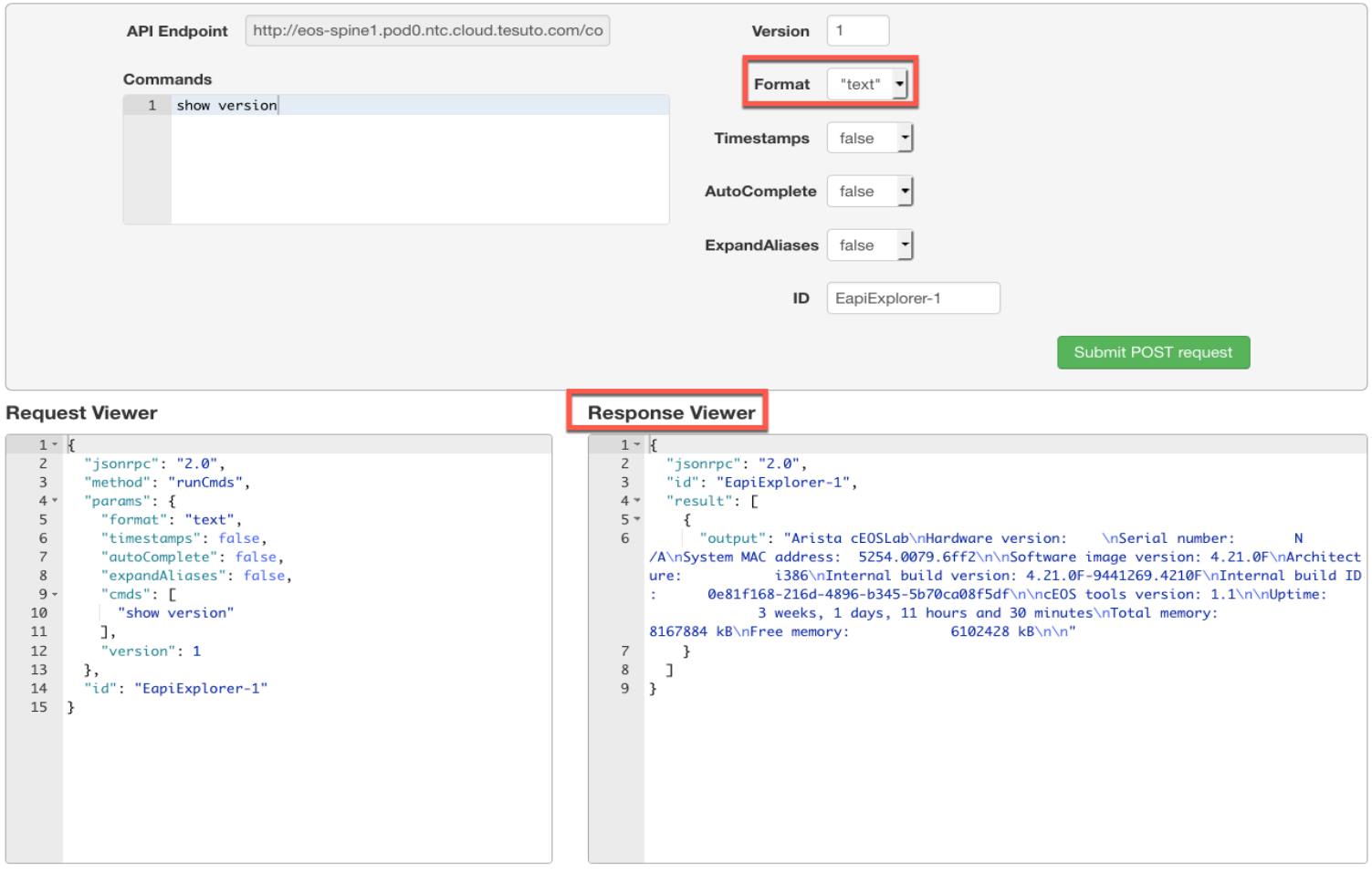
eAPI Command Explorer



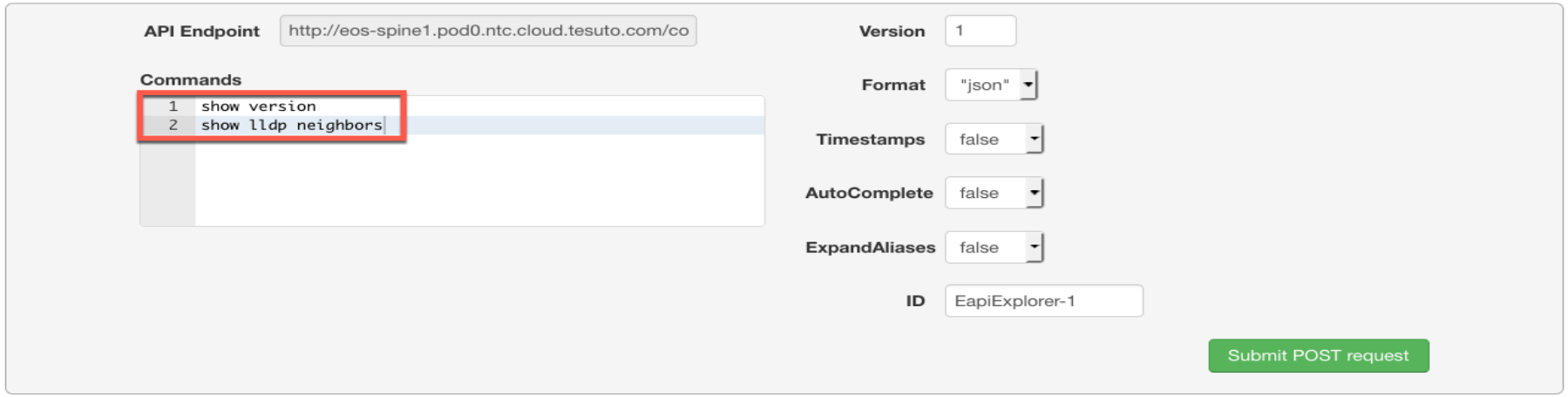
JSON Response



Text Response



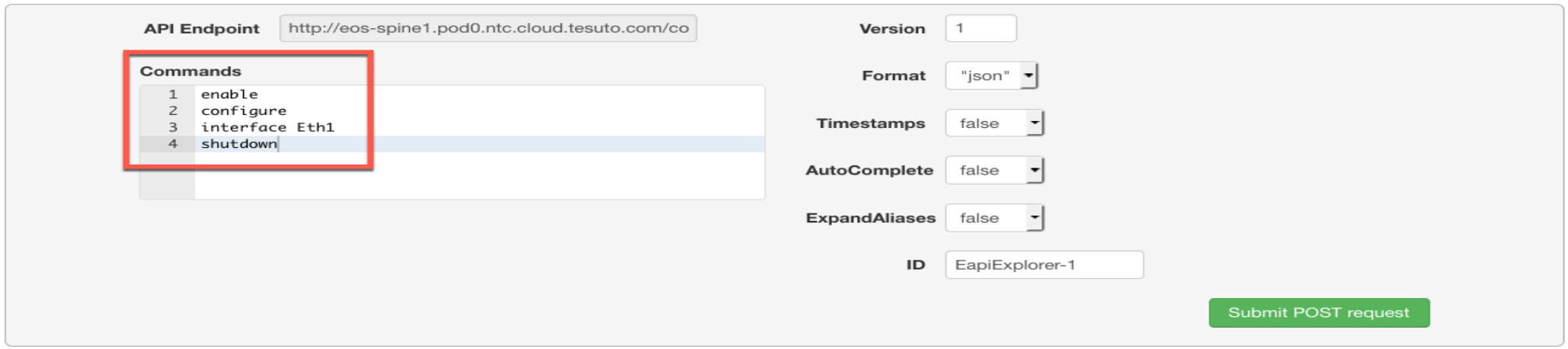
Sending Multiple Show Commands



Request Viewer

Response Viewer

Sending Multiple Config Commands



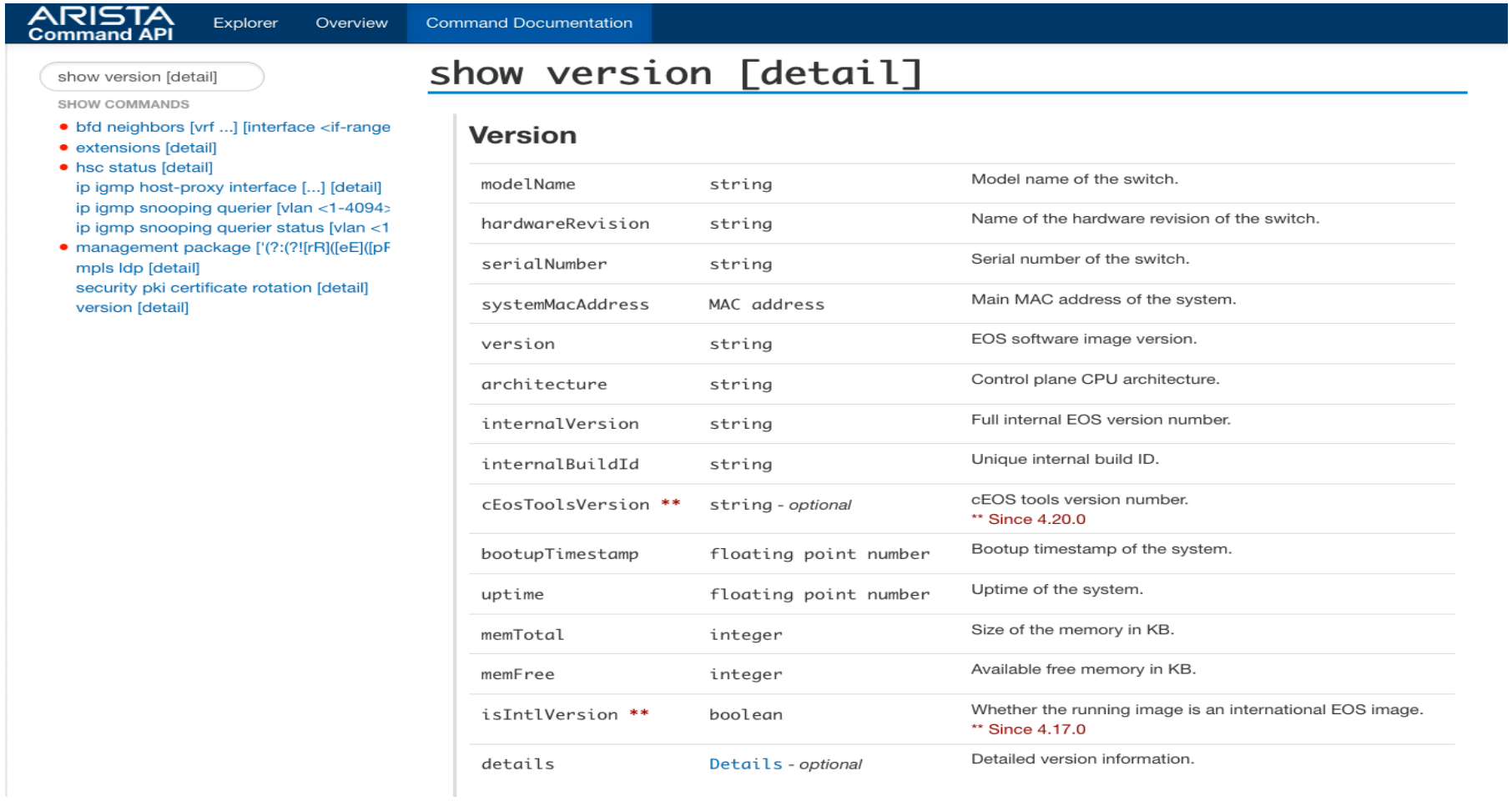
Request Viewer

```
{
  "jsonrpc": "2.0",
  "method": "runCmds",
  "params": {
    "format": "json",
    "timestamps": false,
    "autoComplete": false,
    "expandAliases": false,
    "cmds": [
      "enable",
      "configure",
      "interface Eth1",
      "shutdown"
    ],
    "version": 1
  },
  "id": "EapiExplorer-1"
}
```

Response Viewer

```
{
  "jsonrpc": "2.0",
  "id": "EapiExplorer-1",
  "result": [
    {},
    {},
    {},
    {}
  ]
}
```

Command Documentation



Demo

- Cisco Nexus NX-API Sandbox
- Arista eAPI Command Explorer

Note: these are learning and testing tools.

Lab Time

- Lab 22 - Exploring eAPI and NXAPI
  - Lab 22.1 - Exploring Arista eAPI
  - Lab 22.2 - Exploring Cisco NXAPI

Choose either the eAPI Command Explorer or NX-API Developer Sandbox Lab

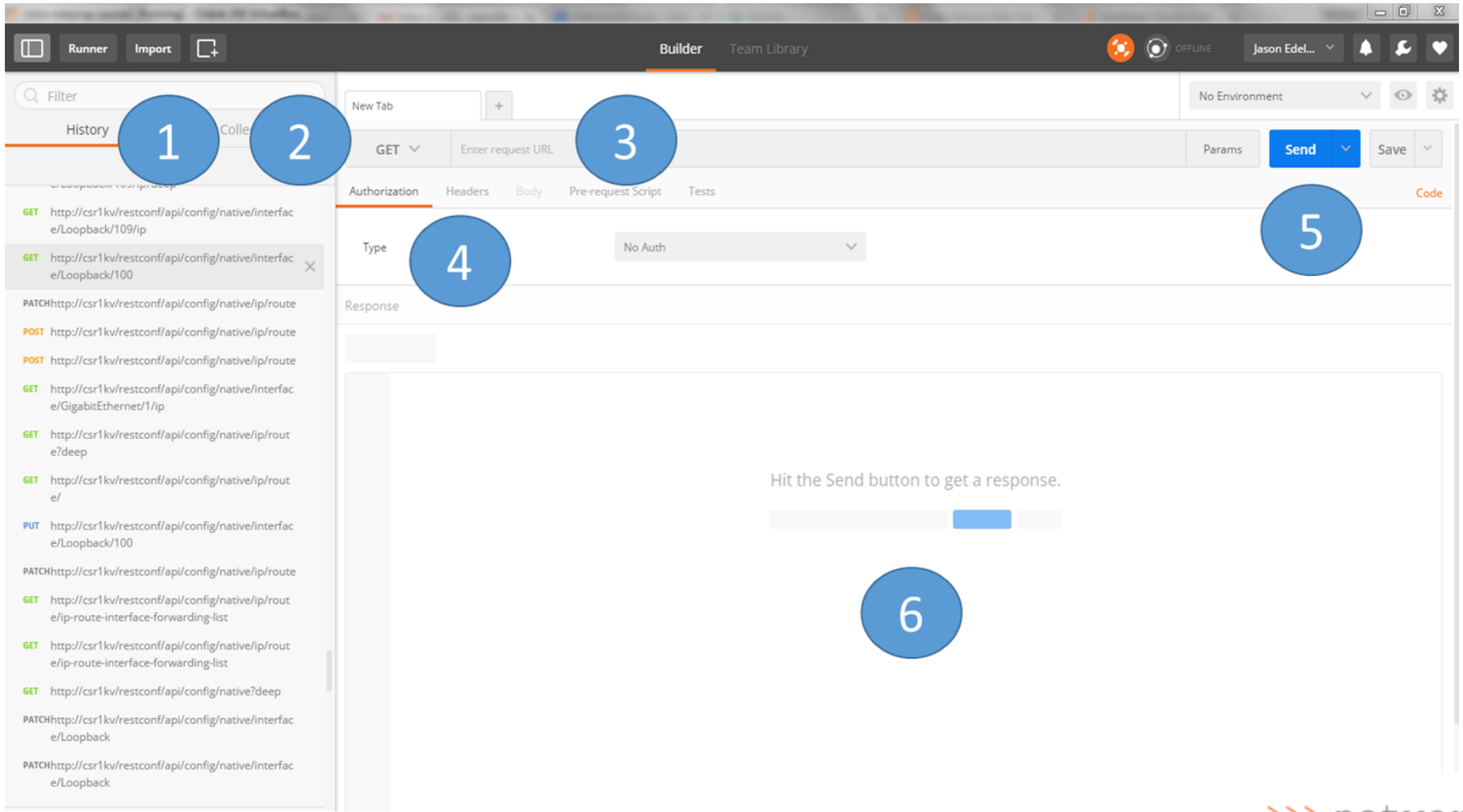
Learning how to use HTTP APIs with Postman

Network APIs

Postman

- User intuitive GUI application to interact with HTTP-based APIs.
- Primarily Used for testing and learning
- You can create a job collection

Postman (cont'd)



RESTful HTTP APIs

Network APIs

RESTful APIs

- URL maps to a particular resource
- HTTP Request Type specifies requested operation

We are going to look at the IOS-XE RESTCONF API.

What is RESTCONF?

- Functional sub-set of NETCONF
- Exposes YANG models via a REST API (URL)
- Uses HTTP(S) as transport
- Uses XML or JSON for encoding
- Uses standard HTTP verbs in REST APIs
- Content-Type & Accept Headers:
  - application/yang-data+json
  - application/yang-data+xml

Note: Must exit configuration mode after making a change for it to be readable via RESTCONF

RESTCONF on IOS-XE

Enabling RESTCONF

```
restconf
!
username <username> privilege 15 password <password>
!
ip http server
ip http secure-server
!
```

Request Methods

GET - Retrieves data from the specified object

PUT - Replaces full configuration object of tree specified

POST- Creates the object with the supplied information

DELETE - Deletes the specified object

PATCH - Applies partial modifications to the specified object

RESTCONF Example 1

Retrieve a full running configuration modeled as JSON.

```
Method: GET
URL: 'http://csr1/restconf/data/Cisco-IOS-XE-native:native?cor
Accept-Type: application/yang-data+json
```


```
{
#output removed for example
"interface": {
            "GigabitEthernet": [
                {
                    "name": "1",
                    "description": "MANAGEMENT_INTEFACE__DO_NOT_CHANGE",
                    "ip": {
                        "address": {
                            "dhcp": {}
                        }
                    },
                    "mop": {
                        "enabled": false,
                        "sysid": false
                    },
                    "Cisco-IOS-XE-cdp:cdp": {
                        "enable": true
                    },
                    "Cisco-IOS-XE-ethernet:negotiation": {
                        "auto": true
                    }
                },
                {
                    "name": "10",
                    "shutdown": [
                        null
                    ],
                    "ip": {
                        "no-address": {
                            "address": false,
                            #output removed for example
                        }
                    }
                }
            ]
        }
    }
```

RESTCONF Example 2

The depth-query parameter is used to limit the depth of subtrees returned by the server.

```
Method: GET
URL: 'http://csr1/restconf/data/Cisco-IOS-XE-native:native?cor
Accept-Type: application/yang-data+json
```

- The value of the "depth" parameter is either an integer between 1 and 65535 or the string "unbounded"
- If not present in URI, the default value is: “unbounded”
- Only allowed for GET/HEAD method


```
{
#output removed for example
"interface": {
            "GigabitEthernet": [
                {
                    "name": "1",
                    "description": "MANAGEMENT_INTEFACE__DO_NOT_CHANGE",
                    "ip": {},
                    "mop": {},
                    "Cisco-IOS-XE-cdp:cdp": {},
                    "Cisco-IOS-XE-ethernet:negotiation": {}
                },
                {
                    "name": "10",
                    "shutdown": [
                        null
                    ],
                    "ip": {},
                    "mop": {},
                    "Cisco-IOS-XE-ethernet:negotiation": {}
                },
                {
                    "name": "11",
                    "shutdown": [
                        null
                    ],
                    "ip": {},
                    "mop": {},
                    "Cisco-IOS-XE-ethernet:negotiation": {}
                }
              ]
            }
          }
```

RESTCONF Example 2

Narrowing the scope and examining the hierarchy

```
{
  "Cisco-IOS-XE-native:GigabitEthernet": {
    "GigabitEthernet": [
      {
        "name": "3",
        "ip": {
          "address": {
            "primary": {
              "address": "10.2.0.151",
              "mask": "255.255.255.0"
            }
          }
        }
      }
    ]
  }
}
#output ommited
```

Pattern

Cisco-IOS-XE-native:GigabitEthernet (dict) -> GigabitEthernet (list) -> ip (dict) -> address (dict) -> primary (dict)

RESTCONF Example 3

Request:

```
Method: GET
URL: 'http://csr1/restconf/data/Cisco-IOS-XE-native:native/interface/GigabitEthernet=3/ip'
Accept-Type: application/yang-data+json
```

Response:

```
{
    "Cisco-IOS-XE-native:ip": {
        "address": {
            "primary": {
                "address": "10.2.0.151",
                "mask": "255.255.255.0"
            }
        }
    }
}
```

RESTCONF Example 4

Understanding PUT, PATCH, POST by Updating an Interface

Existing Configuration

```
interface Loopback100
 ip address 222.22.2.2 255.255.255.0 secondary
 ip address 100.2.2.2 255.255.255.0
```

BODY Used for POST, PATCH, PUT:

```
{
    "Cisco-IOS-XE-native:Loopback": {
        "name": 100,
        "ip": {
            "address": {
                "primary": {
                    "address": "100.2.2.2",
                    "mask": "255.255.255.0"
                }
            }
        }
    }
}
```

RESTCONF Example 4 - The Result

Request 1:

POST http://csr1/restconf/data/Cisco-IOS-XE-native:native/interface/

Response: 409; Error: Object Already Exists; No change in config

RESTCONF Example 4 - The Result

Request 1:

POST http://csr1/restconf/data/Cisco-IOS-XE-native:native/interface/

Response: 409; Error: Object Already Exists; No change in config

Request 2:

PATCH http://csr1/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback

Response 204; No change in config

RESTCONF Example 4 - The Result

Request 1:

POST http://csr1/restconf/data/Cisco-IOS-XE-native:native/interface/

Response: 409; Error: Object Already Exists; No change in config

Request 2:

PATCH http://csr1/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback

Response 204; No change in config

Request 3:

PUT http://csr1/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100

Response 204;

RESTCONF Example 4 - The Result

RESULT FOR THE PUT

Existing Configuration

```
interface Loopback100
 ip address 100.2.2.2 255.255.255.0
```

Static Route Management

Using RESTCONF to manage static route configuration

Starting Configuration:

```
csr1# show run | inc route
 ip route 0.0.0.0 0.0.0.0 10.0.0.2
```

RESTCONF Example 5 - PATCHing Routes

PATCH http://csr1/restconf/data/Cisco-IOS-XE-native:native/ip/route

Body:

```
{
    "Cisco-IOS-XE-native:route": {
        "ip-route-interface-forwarding-list":[
        {
           "prefix":"172.16.0.0",
           "mask":"255.255.0.0",
           "fwd-list":[
              {
                 "fwd":"192.168.1.1"
              }
           ]
        },
        {
           "prefix":"10.0.100.0",
           "mask":"255.255.255.0",
           "fwd-list":[
              {
                 "fwd":"192.168.1.1"
              }
           ]
        }
     ]
   }
}
```

RESTCONF Example 5 - PATCHing Routes (cont'd)

Resulting New Configuration:

```
csr1# show run | inc route
ip route 0.0.0.0 0.0.0.0 10.0.0.2
ip route 10.0.100.0 255.255.255.0 192.168.1.1
ip route 172.16.0.0 255.255.0.0 192.168.1.1
```

RESTCONF Example 6 - PUTing Routes

Starting Configuration:

```
csr1#show run | inc route
ip route 0.0.0.0 0.0.0.0 10.0.0.51
ip route 10.0.100.0 255.255.255.0 192.168.1.1
ip route 172.16.0.0 255.255.0.0 192.168.1.1
```

RESTCONF Example 6 - PUTing Routes (cont'd)

PUT http://csr1/restconf/data/Cisco-IOS-XE-native:native/ip/route

Body:

```
{
   "Cisco-IOS-XE-native:route": {
      "ip-route-interface-forwarding-list":[
         {
            "prefix":"0.0.0.0",
            "mask":"0.0.0.0",
            "fwd-list":[
               {
                  "fwd":"10.0.0.2"
               }
            ]
         }
      ]
   }
}
```

RESTCONF Example 6 - PUTing Routes (cont'd)

Resulting New Configuration:

```
csr1# show run | inc route
ip route 0.0.0.0 0.0.0.0 10.0.0.2
```

Summary

- True REST APIs are powerful
- Be careful using PUTs
- With great power comes great responsibility

Lab Time

- Lab 23 - Exploring Postman
  - Lab 23.1 Exploring IOS-XE RESTCONF API
  - Lab 23.2 Exploring Arista eAPI

Note: Feel free to test it with IOS-XE RESTCONF or eAPI.

Bonus Material

Consuming HTTP APIs with Python requests

Network APIs

Python requests

- Python module to interact with HTTP based APIs (REST)
- Useful functions are **post** and **get**
  - Function per HTTP verb, i.e. **post** is used for POST requests and **get** is used for GET requests
- Optional, helper method for basic Authentication
- Headers used to dictate data encoding

```
import requests
from requests.auth import HTTPBasicAuth

auth = HTTPBasicAuth('ntc', 'ntc123')

headers = {
    'Content-Type': 'application/json',
    'Accept': 'application/json'
}
```

Sample GET:

```
response = requests.get('http://<device>', headers=headers, auth=auth)
```

Python requests

- `data` must be a JSON string - must use `json.dumps()`
- `data`, `headers`, and `auth` are defined parameters that must be used within the requests library
- `payload` is an arbitrary variable that maps back to device API requirements

```
import requests
import json
from requests.auth import HTTPBasicAuth

auth = HTTPBasicAuth('ntc', 'ntc123')

headers = {
    'Content-Type': 'application/json',
    'Accept': 'application/json'
}
payload = <# some dictionary #>

url = 'http://eos-spine1/command-api'

response = requests.post(url, data=json.dumps(payload), headers=headers, auth=auth)
```

Python requests

```
#!/usr/bin/env python
import requests
import json
from requests.auth import HTTPBasicAuth

if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    headers = {
        'Content-Type': 'application/json'
    }
    payload = {
        "jsonrpc": "2.0",
        "method": "runCmds",
        "params": {
            "version": 1,
            "cmds": [
                "show version"
            ],
            "format": "json",
            "timestamps": False
        },
        "id": "ntc"
    }
    url = 'http://eos-spine1/command-api'
    response = requests.post(url, data=json.dumps(payload), he
    rx_object = json.loads(response.text)
    print('Status Code: ' + str(response.status_code))
```

Python requests

```
#!/usr/bin/env python
import requests
import json
from requests.auth import HTTPBasicAuth

if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    headers = {
        'Content-Type': 'application/json'
    }
    payload = {
        "jsonrpc": "2.0",
        "method": "runCmds",
        "params": {
            "version": 1,
            "cmds": [
                "show version"
            ],
            "format": "json",
            "timestamps": False
        },
        "id": "ntc"
    }
    url = 'http://eos-spine1/command-api'
    response = requests.post(url, data=json.dumps(payload), he
    rx_object = json.loads(response.text)
    print('Status Code: ' + str(response.status_code))
```

- Run *show version* on a Arista switch
- Print **status_code**, **text** and OS version.
- The **text** attribute contains the response of a request as a JSON string
- The **status_code** attribute contains the HTTP response code

```
Status Code: 200
{
    "jsonrpc": "2.0",
    "result": [
        {
            "memTotal": 3895836,
            "version": "4.15.2F",
            "internalVersion": "4.15.2F-2663444.4152F",
            "serialNumber": "",
            "systemMacAddress": "2c:c2:60:28:54:dd",
            "bootupTimestamp": 1477365548.64,
            "memFree": 1621108,
            "modelName": "vEOS",
            "architecture": "i386",
            "internalBuildId": "0ebbad93-563f-4920-8ecb-731057802b9c",
            "hardwareRevision": ""
        }
    ],
    "id": "ntc"
}
```

Python requests

```
#!/usr/bin/env python
import requests
import json
from requests.auth import HTTPBasicAuth
if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    headers = {
        'Content-Type': 'application/json'
    }
    payload = {
        "jsonrpc": "2.0",
        "method": "runCmds",
        "params": {
            "version": 1,
            "cmds": [
                "show hostname",
                "show vlan"
            ],
            "format": "json",
            "timestamps": False
        },
        "id": "ntc"
    }
    url = 'http://eos-spine1/command-api'
    response = requests.post(url, data=json.dumps(payload), he
    rx_object = json.loads(response.text)
    print('Status Code: ' + str(response.status_code))
    result = rx_object['result']
    print("Hostname: ", json.dumps(result[0], indent=4))
    print("VLANs: ", json.dumps(result[1], indent=4))
```

Python requests

```
#!/usr/bin/env python
import requests
import json
from requests.auth import HTTPBasicAuth
if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    headers = {
        'Content-Type': 'application/json'
    }
    payload = {
        "jsonrpc": "2.0",
        "method": "runCmds",
        "params": {
            "version": 1,
            "cmds": [
                "show hostname",
                "show vlan"
            ],
            "format": "json",
            "timestamps": False
        },
        "id": "ntc"
    }
    url = 'http://eos-spine1/command-api'
    response = requests.post(url, data=json.dumps(payload), he
    rx_object = json.loads(response.text)
    print('Status Code: ' + str(response.status_code))
    result = rx_object['result']
    print("Hostname: ", json.dumps(result[0], indent=4))
    print("VLANs: ", json.dumps(result[1], indent=4))
```

- The **cmds** request parameter is a list.
- Run *show hostname* and *show vlan* at the same time.
- *result* is a list and can be used to print individual command output.

```
Status Code: 200
Hostname:  {
    "hostname": "eos-spine1",
    "fqdn": "eos-spine1.ntc.com"
}
VLANs:  {
    "sourceDetail": "",
    "vlans": {
        "1": {
            "status": "active",
            "interfaces": {},
            "dynamic": false,
            "name": "default"
        }
    }
}
```

Using requests with IOS-XE

```
#!/usr/bin/env python
import requests
import json
from requests.auth import HTTPBasicAuth

if __name__ == "__main__":
    auth = HTTPBasicAuth('ntc', 'ntc123')
    headers = {
        'Accept-Type': 'application/vnd.yang.data+json',
        'Content-Type': 'application/vnd.yang.data+json'
    }

    url = 'http://csr1/restconf/api/config/native/interface'
    response = requests.get(url, headers=headers, auth=auth)
    print('Status Code: ' + str(response.status_code))
    print("\nInterfaces Object: ", response.text)
```

```
Status Code: 200

{
  "ned:interface": {
    "GigabitEthernet": [
      {
        "name": "1"
      },
      {
        "name": "2"
      },
      {
        "name": "3"
      },
      {
        "name": "4"
      }
    ]
  }
}
```

Lab Time

- Lab 24 - Using Python requests:
  - 24.1 - requests using eAPI
  - 24.2 - requests using NX-API

Pick one of the labs.

NAPALM

Network APIs

NAPALM

NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) is a Python library that implements a set of functions to interact with different network device Operating Systems using a unified API.

NAPALM supports several methods to connect to the devices, to manipulate configurations or to retrieve data.

https://napalm.readthedocs.io/en/latest/

NAPALM Support Matrix

- Palo Alto PANOS
- Cisco IOS
- Cisco NX-OS
- Cisco IOS-XR
- Arista EOS
- Juniper Junos
- IBM
- Pluribus
- FortiOS
- Cumulus Linux
- Actively growing

NAPALM

Three core functions:

- Retrieving Data
- Declarative Configuration Management
- Deployment Validation

All three are done in a uniform and vendor-neutral fashion

Retrieving Data

Uses a uniform and consistent data model across all device types supported by NAPALM

- get_facts()
- get_arp_table()
- get_bgp_config()
- get_bgp_neighbors()
- get_bgp_neighbors_detail()
- get_interfaces()
- get_interfaces_counters()
- get_lldp_neighbors()
- get_lldp_neighbors_detail()
- get_ntp_peers()
- get_ntp_stats()
- get_ntp_servers()
- ... plus another dozen and growing...

get_facts()

Network Device Facts

```
>>> device.get_facts()
{'os_version': u'4.15.2F-2663444.4152F', 'uptime': 5817, 'interface_list': [u'Ethernet1', u'Ethernet2', u'Ethernet3',
u'Ethernet4', u'Ethernet5', u'Ethernet6', u'Ethernet7', u'Management1'], 'vendor': u'Arista', 'serial_number': u'', 'model':
u'vEOS', 'hostname': u'eos-spine1', 'fqdn': u'eos-spine1.ntc.com'}
>>>
>>> facts = device.get_facts()
>>>
>>> import json
>>>
>>> print(json.dumps(facts, indent=4))
{
    "os_version": "4.15.2F-2663444.4152F",
    "uptime": 5837,
    "interface_list": [
        "Ethernet1",
        "Ethernet2",
        "Ethernet3",
        "Ethernet4",
        "Ethernet5",
        "Ethernet6",
        "Ethernet7",
        "Management1"
    ],
    "vendor": "Arista",
    "serial_number": "",
    "model": "vEOS",
    "hostname": "eos-spine1",
    "fqdn": "eos-spine1.ntc.com"
}
>>>
```

get_interfaces()

Gathering Interfaces Info

```
>>> device.get_interfaces()
{u'Management1': {'is_enabled': True, 'description': u'', 'last_flapped': 1467419703.021217, 'is_up': True, 'mac_address':
u'2c:c2:60:0d:52:90', 'speed': 1000}, u'Ethernet2': {'is_enabled': True, 'description': u'', 'last_flapped': 1467419702.781202,
'is_up': True, 'mac_address': u'2c:c2:60:12:98:52', 'speed': 1000}, u'Ethernet3': {'is_enabled': True, 'description': u'',
'last_flapped': 1467419702.781203, 'is_up': True, 'mac_address': u'2c:c2:60:60:20:9b', 'speed': 1000}, u'Ethernet1':
{'is_enabled': True, 'description': u'', 'last_flapped': 1467419703.1052225, 'is_up': True, 'mac_address': u'2c:c2:60:2d:45:d5',
'speed': 1000}, u'Ethernet6': {'is_enabled': True, 'description': u'', 'last_flapped': 1467419702.781202, 'is_up': True,
'mac_address': u'2c:c2:60:48:80:70', 'speed': 1000}, u'Ethernet7': {'is_enabled': True, 'description': u'', 'last_flapped':
1467419702.8092043, 'is_up': True, 'mac_address': u'2c:c2:60:40:8d:10', 'speed': 1000}, u'Ethernet4': {'is_enabled': True,
'description': u'', 'last_flapped': 1467419702.769202, 'is_up': True, 'mac_address': u'2c:c2:60:2e:c6:f8', 'speed': 1000},
u'Ethernet5': {'is_enabled': True, 'description': u'', 'last_flapped': 1467419703.105223, 'is_up': True, 'mac_address':
u'2c:c2:60:60:7d:ba', 'speed': 1000}}
>>>
```

get_interfaces() (cont'd)

```
>>> interfaces = device.get_interfaces()
>>>
>>> print(json.dumps(interfaces, indent=4))
{
    "Management1": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419703.0212176,
        "is_up": true,
        "mac_address": "2c:c2:60:0d:52:90",
        "speed": 1000
    },
    "Ethernet2": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419702.7812023,
        "is_up": true,
        "mac_address": "2c:c2:60:12:98:52",
        "speed": 1000
    },
    "Ethernet3": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419702.7812028,
        "is_up": true,
        "mac_address": "2c:c2:60:60:20:9b",
        "speed": 1000
    },
    "Ethernet1": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419702.781203,
        "is_up": true,
        "mac_address": "2c:c2:60:48:80:70",
        "speed": 1000
    },
    "Ethernet5": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419702.8092043,
        "is_up": true,
        "mac_address": "2c:c2:60:40:8d:10",
        "speed": 1000
    },
    "Ethernet4": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419702.7692015,
        "is_up": true,
        "mac_address": "2c:c2:60:2e:c6:f8",
        "speed": 1000
    }
}
```

get_interfaces_ip()

Get Interfaces IP Addresses

```
>>> {u'Management1': {u'ipv4': {u'10.0.0.11': {u'prefix_length': 24}}, u'ipv6': {}}}
{u'Management1': {u'ipv4': {u'10.0.0.11': {u'prefix_length': 24}}, u'ipv6': {}}}
>>>
```

get_environment()

Device Environment Status

```
>>> device.get_environment()
{u'fans': {}, u'memory': {u'available_ram': 99060, u'used_ram': 1798476}, u'temperature': {}, u'power': {}, u'cpu': {0:
{u'%usage': 5.4}}}
>>>
```

NAPALM Configuration Management

Two main ways to manage device configurations with NAPALM

Configuration Replace

- Declarative configuration always pushing the full configuration
- Only commands required to get the device into its intended state are applied
- No "negation (no)" commands are sent to the device

Configuration Merge

- Send a set of commands or configuration stanza
- Only commands required to get the device into its intended state are applied
- You can use the merge for declarative management on a stanza based on OS

It does vary based on operating system.

NAPALM Configuration Management

Example Workflow

Works slightly different than based on individual drivers and operating systems.

1. Connect to Device
2. Copy desired configuration to device (checkpoint file, candidate configuration, config session, bootflash as candidate_config.txt)
3. Use a vendor command to view diffs
4. Use a vendor command to apply configuration changes
5. Optionally, rollback to a config that exists in the file system.

Note: you dictate if the supplied configuration is a full config file or partial configuration

Configuration Replace

Focus on desired configuration commands.

Scenario: You need to remove two loopback interfaces and change the hostname.

NAPALM Config Replace:

- Full configuration is sent to the device, but...
- Only diffs are applied.
- You do not need to worry about going from A to B - you just focus on B.

```
$ more diffs/csr1.diffs
+hostname csr1
-hostname csr_old_name
-interface Loopback100
 -ip address 1.1.1.1 255.255.255.255
-interface Loopback200
 -ip address 22.2.1.1 255.255.255.255
-ip route 10.1.1.0 255.255.255.0 192.0.1.1
```

IMPORTANT: There are no `no` commands used. The underlying OS generates the diffs (for most NAPALM drivers).

Configuration Merge

You can use NAPALM for declarative management for a sectional config too.

Current BGP Config

```
router bgp 65512
   neighbor 10.0.0.0 remote-as 65500
   neighbor 10.0.0.0 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   network 20.20.20.0/24
!
```

Desired BGP Config (file sent to device)

```
router bgp 65512
   neighbor 10.0.0.2 remote-as 65500
   neighbor 10.0.0.2 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   neighbor 10.0.0.10 remote-as 65512
   network 100.0.100.0/24
!
```

Configuration Merge

You can use NAPALM for declarative management for a sectional config too.

Current BGP Config

```
router bgp 65512
   neighbor 10.0.0.0 remote-as 65500
   neighbor 10.0.0.0 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   network 20.20.20.0/24
!
```

Desired BGP Config (file sent to device)

```
router bgp 65512
   neighbor 10.0.0.2 remote-as 65500
   neighbor 10.0.0.2 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   neighbor 10.0.0.10 remote-as 65512
   network 100.0.100.0/24
!
```

Diff Generated by NAPALM

```
     neighbor 10.0.0.0 maximum-routes 12000
     neighbor 10.0.0.1 remote-as 65512
     neighbor 10.0.0.1 maximum-routes 12000
+    neighbor 10.0.0.2 remote-as 65500
+    neighbor 10.0.0.2 maximum-routes 12000
+    neighbor 10.0.0.10 remote-as 65512
+    neighbor 10.0.0.10 maximum-routes 12000
     network 20.20.20.0/24
+    network 100.0.100.0/24
 !
 management api http-commands
    protocol http
```

Configuration Merge (Advanced)

You can use NAPALM for declarative management for a sectional config too.

Current BGP Config

```
router bgp 65512
   neighbor 10.0.0.0 remote-as 65500
   neighbor 10.0.0.0 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   network 20.20.20.0/24
!
```

Desired BGP Config (file sent to device)

```
no router bgp 65512
router bgp 65512
   neighbor 10.0.0.2 remote-as 65500
   neighbor 10.0.0.2 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   neighbor 10.0.0.10 remote-as 65512
   network 100.0.100.0/24
!
```

Configuration Merge (Advanced)

You can use NAPALM for declarative management for a sectional config too.

Current BGP Config

```
router bgp 65512
   neighbor 10.0.0.0 remote-as 65500
   neighbor 10.0.0.0 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   network 20.20.20.0/24
!
```

Desired BGP Config (file sent to device)

```
no router bgp 65512
router bgp 65512
   neighbor 10.0.0.2 remote-as 65500
   neighbor 10.0.0.2 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   neighbor 10.0.0.10 remote-as 65512
   network 100.0.100.0/24
!
```

Diff Generated by NAPALM

```
 router bgp 65512
-   neighbor 10.0.0.0 remote-as 65500
-   neighbor 10.0.0.0 maximum-routes 12000
    neighbor 10.0.0.1 remote-as 65512
    neighbor 10.0.0.1 maximum-routes 12000
-   network 20.20.20.0/24
+   neighbor 10.0.0.2 remote-as 65500
+   neighbor 10.0.0.2 maximum-routes 12000
+   neighbor 10.0.0.10 remote-as 65512
+   neighbor 10.0.0.10 maximum-routes 12000
+   network 100.0.100.0/24
 !
```

Be cautious of device support. This is based on NAPALM driver implementation which is dictated by vendor OS support. This example is EOS.

Getting Started with NAPALM

Step 1. Import Network Driver

```
>>> from napalm import get_network_driver
>>>
>>> driver = get_network_driver('eos')
>>>
```

Step 2. Create Device Object

```
>>> hostname = 'eos-spine1'
>>> username = 'ntc'
>>> password = 'ntc123'
>>>
>>> device = driver(hostname, username, password)
>>>
```

Step 3. Open connection

```
>>> device.open()
>>>
```

Perform a Configuration Merge

Sample new config we want to send/merge with current configuration:

`snmp.conf`

```
snmp-server community networktocode ro
snmp-server community public ro
snmp-server community private rw
snmp-server community supersecret rw
snmp-server location SYDNEY
snmp-server contact JOHN_SMITH
```

Perform a Configuration Merge

- `load_merge_candidate` method
- Support configuration files (`filename` parameter) and strings (`config` parameter)

```
>>> device.load_merge_candidate(filename='snmp.conf')
```

- Compare the running configuration and the new candidate configuration with `compare_config`

```
>>> diffs = device.compare_config()
>>>
>>> print(diffs)
@@ -7,7 +7,12 @@
 hostname eos-spine1
 ip domain-name ntc.com
 !
+snmp-server contact JOHN_SMITH
+snmp-server location SYDNEY
 snmp-server community networktocode ro
+snmp-server community private rw
+snmp-server community public ro
+snmp-server community supersecret rw
 !
 spanning-tree mode mstp
 !
```

Perform a Configuration Replace

- Declarative network configuration management
- Requires using a full configuration file
- `load_replace_candidate` method
- Copies new config to the device, but does not commit it

```
>>> device.load_replace_candidate(filename='new_good.conf')
>>>
```

Other Methods

Method	Description	Example
discard_config	Removes the loaded candidate configuration	device.discard_config()
commit_config	Commit the loaded candidate configuration	device.commit_config()
rollback	Restores a backup configuration saved before the last changes and commit	device.rollback()

```
>>> device.discard_config()
>>>
>>> print(device.compare_config())
u''
```

```
>>> device.commit_config()
>>>
```

```
>>> device.rollback()
>>>
```

How NAPALM Works

- EOS
  - Creates and locks config sessions
  - Uses `rollback clean-config` to prepare for a config replace
  - Commit is performed issuing `copy startup-config flash:rollback-0` , `configure session #` and `commit`
  - Rollback is performed issuing `configure replace flash:rollback-0`
  - Diffs are generated on the device using the `show session-config named <file> diffs`
- IOS
  - Uses SCP or Netmiko (TCL) to transfer config files for config replace/merge
  - Uses `show archive config differences <base_file> <new_file>` to show diffs for config replace
  - Uses `show archive config incremental-diffs <file> ignorecase` to show incremental diffs
  - Replaces with `configure replace <file> force` . Merges with `copy <file> running-config`

How NAPALM Works

- Junos
  - Uses junos-pyez API with NETCONF Junos candidate configurations
  - Locks configurations while performing operations till first commit/rollback
  - Uses `rollback 0` to rollback configuration
- NXOS
  - Uses checkpoint files for config replacement. A checkpoint file can be obtained with `device._get_checkpoint_file()` which issues `checkpoint file temp_cp_file_from_napalm` on the device and then prints it
  - Diffs for config replacement are a list of commands that would be needed to take the device from its current state to the desired config state using `show diff rollback-patch file <source_of_truth_file> file <config_file>` command
  - Merges send config line by line. This doesn't use the checkpoint/rollback functionality. As a result, merges are not atomic
  - Replaces uses `rollback running file <config_file>` command

Lab Time - BONUS

- Lab 25 - NAPALM
  - Using the NAPALM Python Library to do declarative config merge, full config merge and getters for Arista EOS
  - Using the NAPALM Python Library to do basic config merge and getters for Cisco IOS
  - Using the NAPALM Python Library to do declarative config merge, full config merge and getters for Juniper JUNOS