
CNIT-381
FALL 2020

NETCONF-YANG

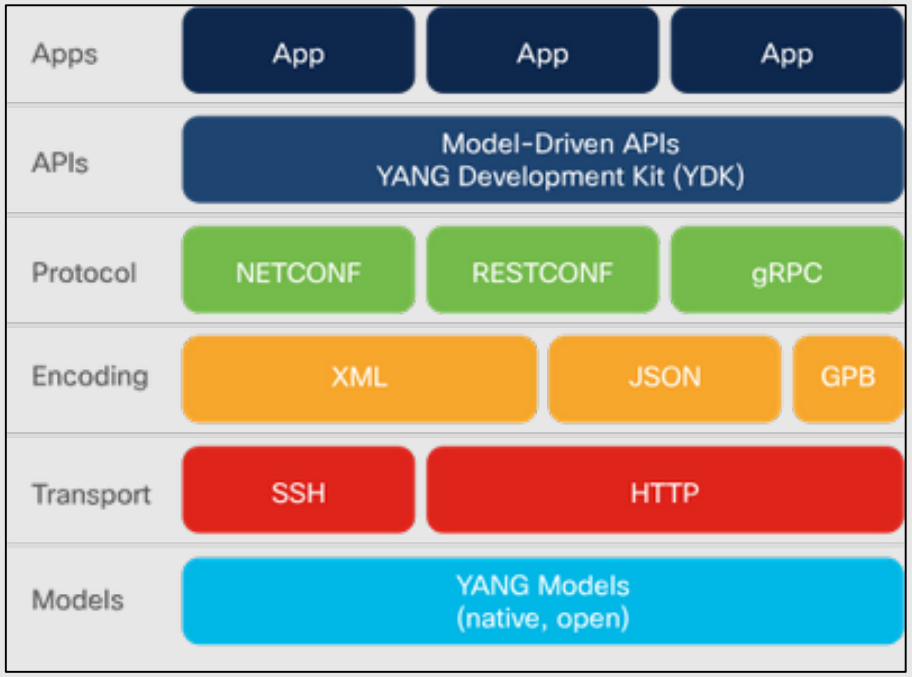
Agent based / Model-Driven API

What is Model-Driven Programmability?

Model-driven programmability:

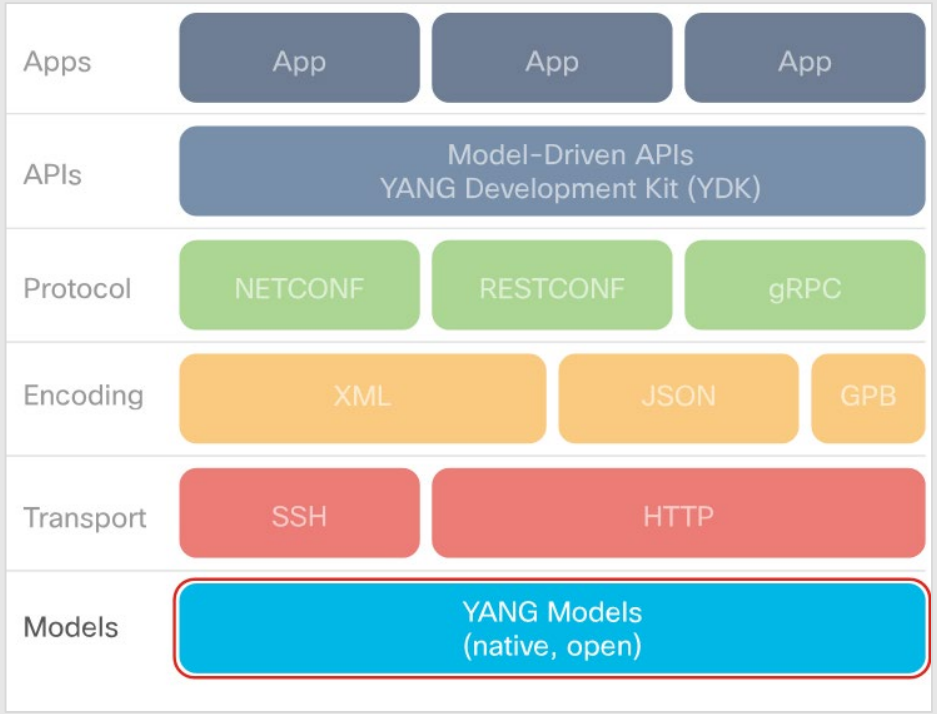
- Provides configuration language that is human-readable
- Is Model-based, structured, and computer-friendly
- Includes support for multiple model types, including native, OpenConfig, and IETF
- Uses specification that is decoupled from transport, protocol end encoding.
- Uses model-driven APIs for abstraction and simplification.
- Leverages open-source and enjoys wide support.

Yang-based Model-driven Programmability Stack



What is YANG?

- Yet Another Next Generation, (YANG) as defined in RFC7519, is "a data modeling language used to model configuration and state data manipulated by the Network Configuration Protocol (NETCONF), NETCONF remote procedure calls, and NETCONF notifications."
- In the real world, there are two types of YANG models, open and native.
- YANG allows different network device vendors to describe their device type, configuration, and state to map to the device operation in programmatic way.

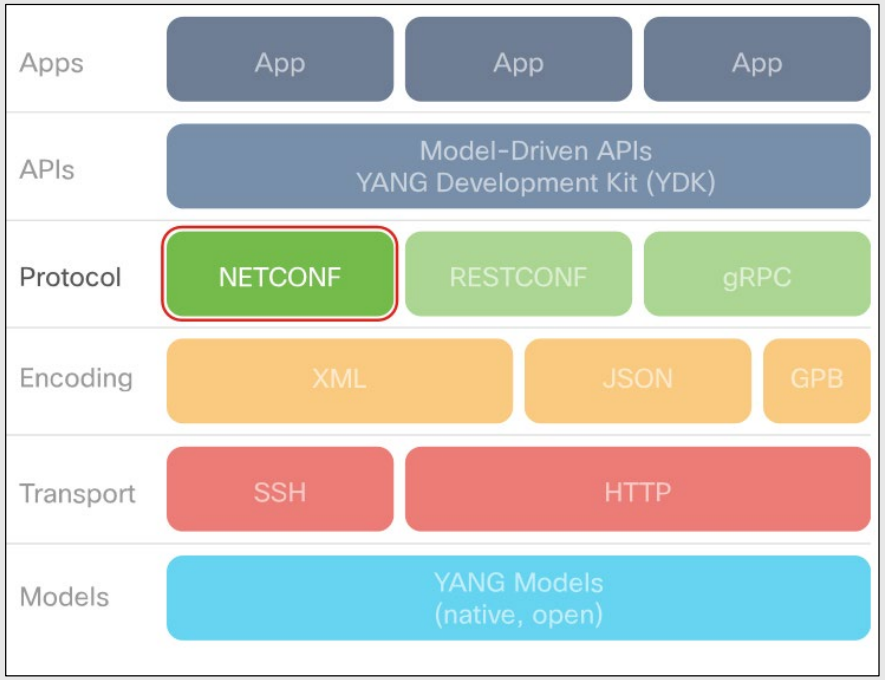


What is YANG? (Cont.)

```
module: ietf-interfaces
   +--rw interfaces
   |  +--rw interface* [name]
   |     +--rw name                        string
   |     +--rw description?                string
   |     +--rw type                        identityref
   |     +--rw enabled?                    boolean
   |     +--rw link-up-down-trap-enable?   enumeration {if-mib}?
   +--ro interfaces-state
      +--ro interface* [name]
         +--ro name               string
         +--ro type               identityref
         +--ro admin-status       enumeration {if-mib}?
         +--ro oper-status        enumeration
         +--ro last-change?       yang:date-and-time
         +--ro if-index           int32 {if-mib}?
         +--ro phys-address?      yang:phys-address
         +--ro higher-layer-if*   interface-state-ref
         +--ro lower-layer-if*    interface-state-ref
         +--ro speed?             yang:gauge64
         +--ro statistics
            +--ro discontinuity-time    yang:date-and-time
            +--ro in-octets?            yang:counter64
            +--ro in-unicast-pkts?      yang:counter64
            +--ro in-broadcast-pkts?    yang:counter64
            +--ro in-multicast-pkts?    yang:counter64
            +--ro in-discards?          yang:counter32
            +--ro in-errors?            yang:counter32
            +--ro in-unknown-protos?    yang:counter32
            +--ro out-octets?           yang:counter64
            +--ro out-unicast-pkts?     yang:counter64
            +--ro out-broadcast-pkts?   yang:counter64
            +--ro out-multicast-pkts?   yang:counter64
            +--ro out-discards?         yang:counter32
            +--ro out-errors?           yang:counter32
```

What is NETCONF?

- Network Configuration (NETCONF), a protocol defined by the IETF RFC7519, is designed to install, manipulate, and delete the configuration of network devices.
- It is the primary protocol used with YANG data models today.
- The NETCONF protocol uses XML-based data encoding for both the configuration data and the protocol messages.
- It provides a small set of operations to manage device configurations and retrieve device state information.



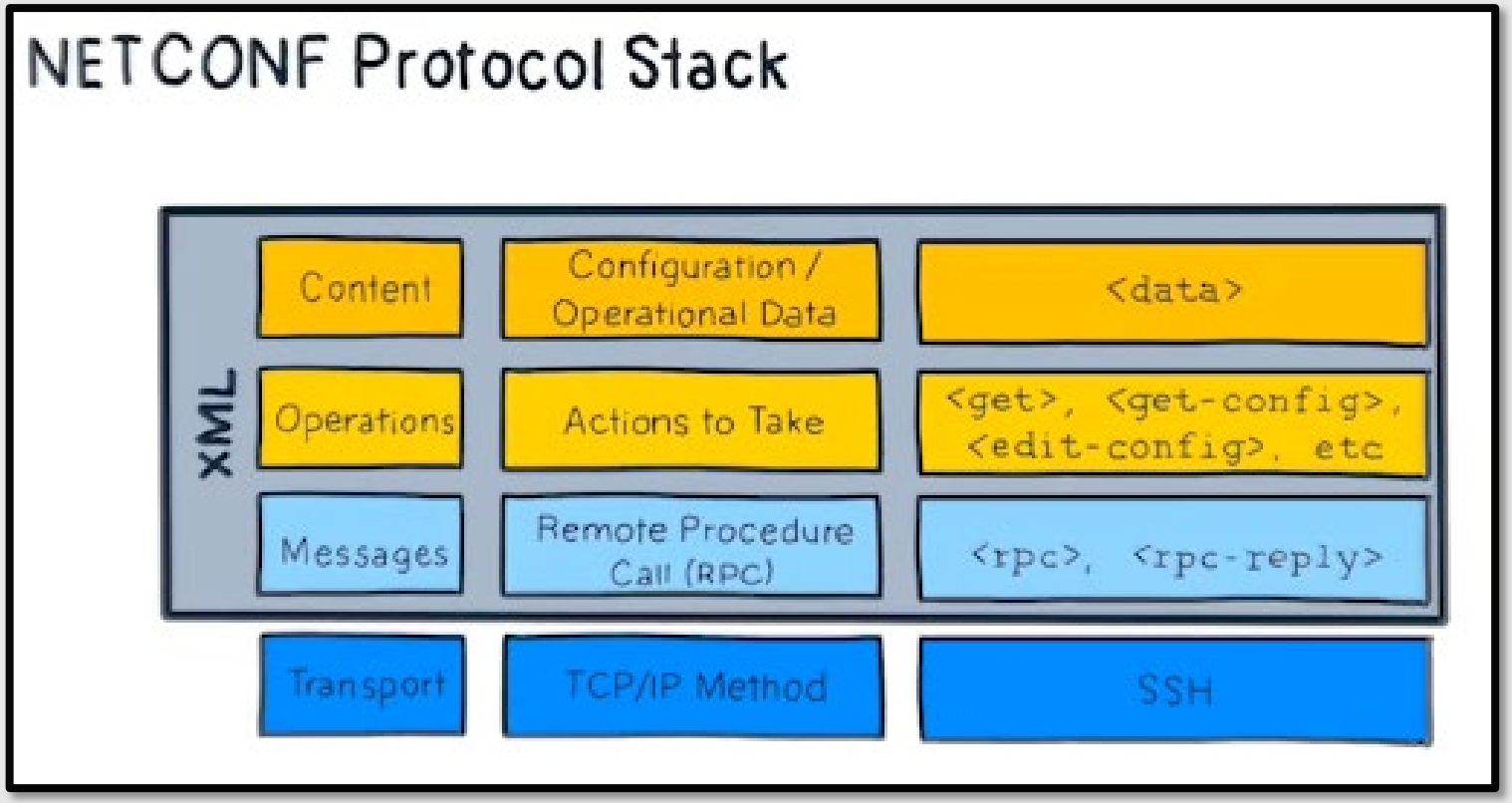
What is NETCONF? (Cont.)

NETCONF Protocol Operations

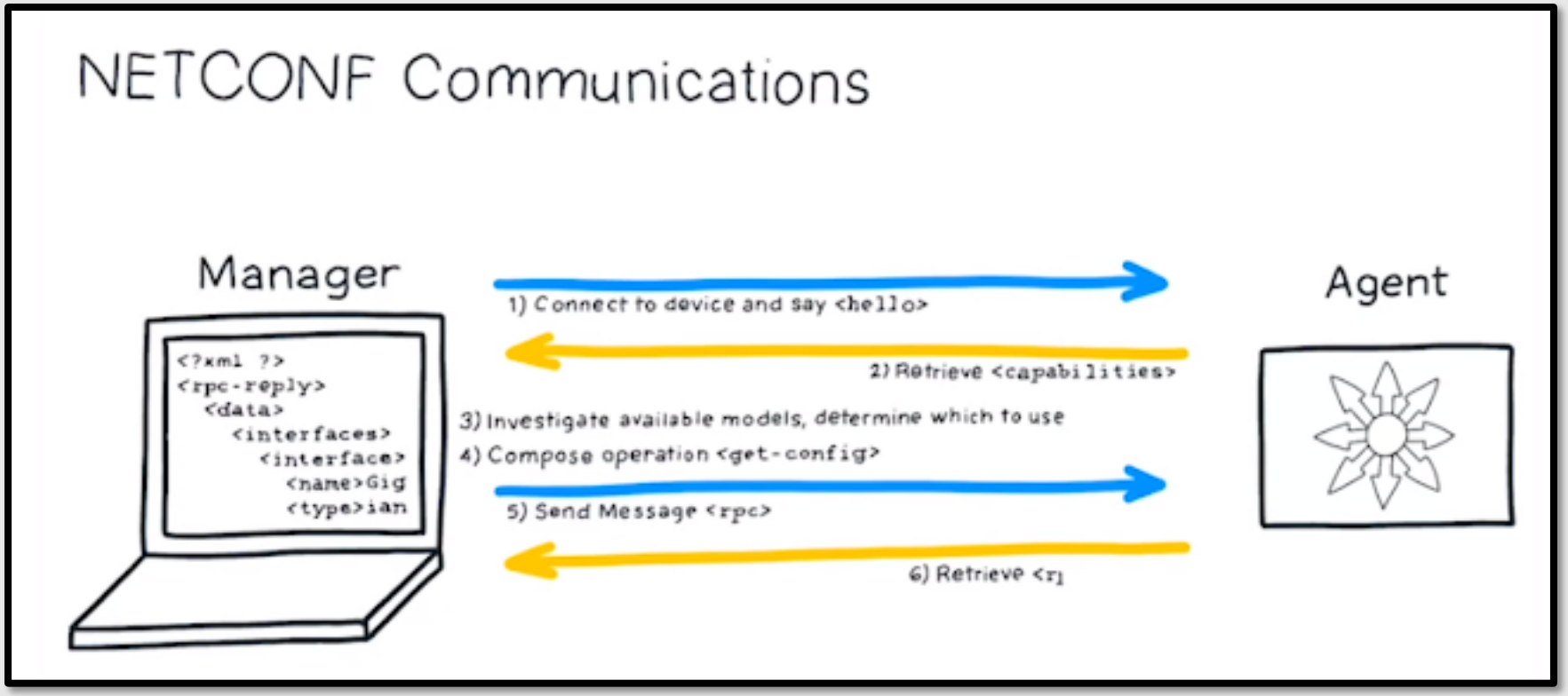
- The NETCONF protocol use RPC calls to provide a set of operations to manage device configurations and retrieve device state information. The base protocol includes the following operations:

NETCONF Protocol Operation		
get	copy-config	unlock
get-config	delete-config	close-session
edit-config	lock	kill-session

What is NETCONF? (Cont.)

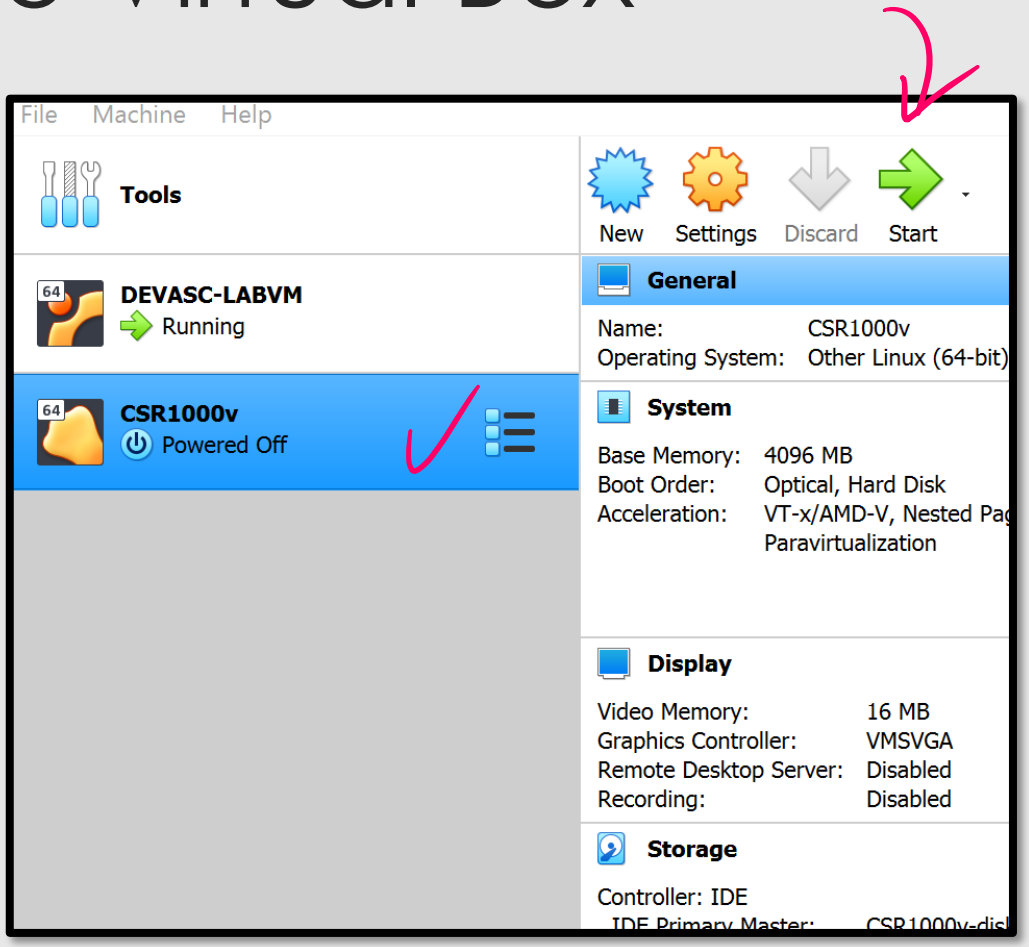


What is NETCONF? (Cont.)

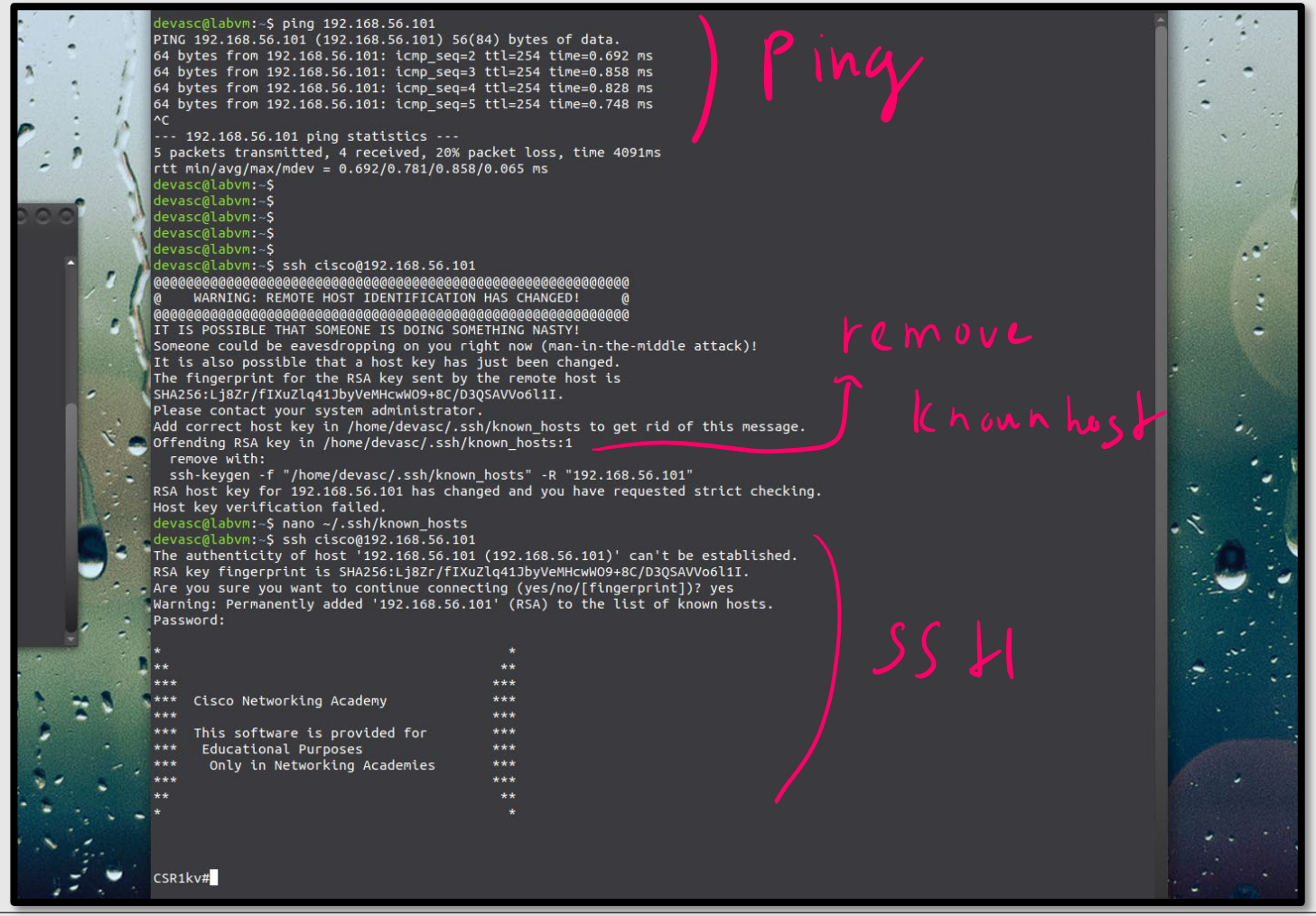


Import IOS-XE Router to Virtual Box

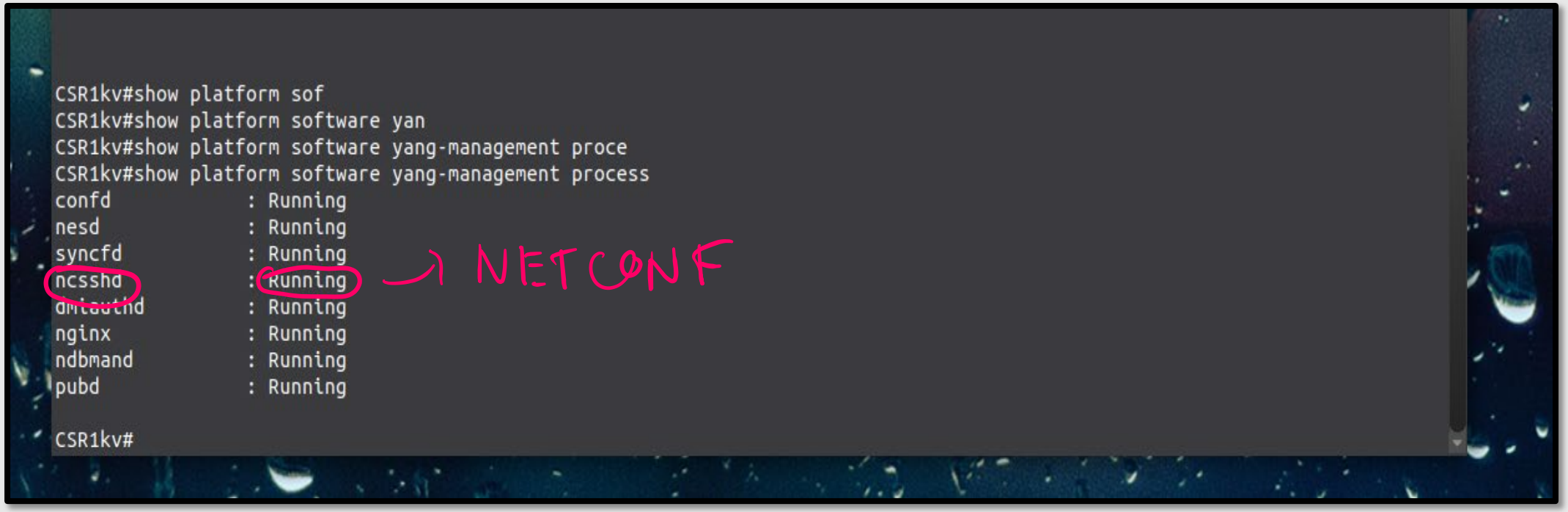
- Open Virtual Box
- Select ***File - > Import Appliance***
- In ***Appliance to Import*** window, select ***source*** as Local System, and navigate your *CSR100v.ova* file in ***File***
- Click ***Next*** then ***Import***
- After finish, you will see your router in virtual box, go ahead and start it



Launch the VMs and Verify Connectivity



Check NETCONF on the CSR1kv



Access the NETCONF process through an SSH terminal

- Enter the following command in a terminal window. You can use the up arrow to recall the latest SSH command and just add the -p and -s options as shown. Then, enter cisco123! as the password.

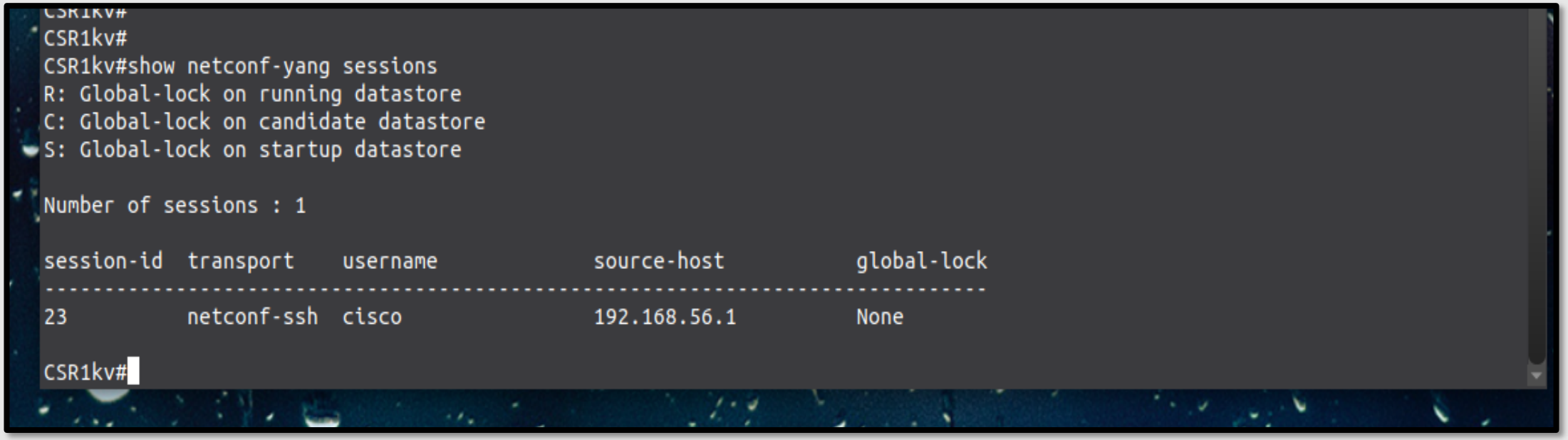
devasc@labvm:~$ ssh cisco@192.168.56.101 -p 830 -s netconf
cisco@192.168.56.101's password:

- The CSR1kv will respond with a hello message that includes over 400 lines of output listing all of its NETCONF capabilities. The end of NETCONF messages are identified with]]>]]>.

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
<capability>urn:ietf:params:netconf:base:1.0</capability>
<capability>urn:ietf:params:netconf:base:1.1</capability>
<capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
<capability>urn:ietf:params:netconf:capability:xpath:1.0</capability>
<capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
<capability>urn:ietf:params:netconf:capability:validate:1.1</capability>
(output omitted)
    </capability>
</capabilities>
<session-id>20</session-id></hello>]]>]]>
```

Sending a hello message from the client

Verify NETCONF Session in CSR1000v



Send RPC messages to an IOS XE device

Copy and paste the following RPC get message XML code into the terminal SSH NETCONF session to retrieve information about the interfaces.

```
<rpc message-id="103" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <get>
  <filter>
    <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"/>
   </filter>
 </get>
</rpc>
]]>]]>
```

Send RPC messages to an IOS XE device

Recall that XML does not require indention or white space. Therefore, the CSR1kv will return a long string of XML data.

<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="103"><data><interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"><interface><name>GigabitEthernet1</name><description>VBox</description><type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift:ethernetCsmacd</type><enabled>true</enabled><ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"></ipv4><ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"></ipv6></interface></interfaces></data></rpc-reply>]]>]]>

Copy the XML that was returned, but do not include the final "]]>]]>" characters. These characters are not part of the XML that is returned by the router.

Search the internet for "prettify XML". Find a suitable site and use its tool to transform your XML into a more readable format

Send RPC messages to an IOS XE device

XML Pretty Printed

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="103">
  <data>
    <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
      <interface>
        <name>GigabitEthernet1</name>
        <description>VBox</description>
        <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift:ethernetCsmacd</type>
        <enabled>true</enabled>
        <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"/>
        <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"/>
      </interface>
    </interfaces>
  </data>
</rpc-reply>
```

USE PYTHON → NETCONF

```python
from ncclient import manager

m = manager.connect(
    host="192.168.56.101",
    port=830,
    username="cisco",
    password="cisco123!",
    hostkey_verify=False
    )

print("#Supported Capabilities (YANG models):")
for capability in m.server_capabilities:
    print(capability)
```

manager → object

port=830 → SSH port 830

.....
urn:ietf:params:xml:ns:yang:smiv2:TUNNEL-MIB?module=TUNNEL-MIB&revision=2005-05-16
urn:ietf:params:xml:ns:yang:smiv2:UDP-MIB?module=UDP-MIB&revision=2005-05-20
urn:ietf:params:xml:ns:yang:smiv2:VPN-TC-STD-MIB?module=VPN-TC-STD-MIB&revision=2005-11-15
urn:ietf:params:xml:ns:netconf:base:1.0?module=ietf-netconf&revision=2011-06-01
urn:ietf:params:xml:ns:yang:ietf-netconf-with-defaults?module=ietf-netconf-with-defaults&revision=2011-06-01

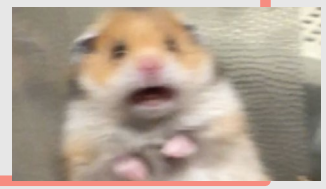
urn:ietf:params:netconf:capability:notification:1.1

USE PYTHON

```python
from ncclient import manager

m = manager.connect(
    host="192.168.56.101",
    port=830,
    username="cisco",
    password="cisco123!",
    hostkey_verify=False
    )

netconf_reply = m.get_config(source="running")
print(netconf_reply)
```

~ Show run

.....
e-external-groups>true</enable-external-groups><rule-list><name>admin</name><group>PRIV15</group><rule><name>permit-all</name><module-name>*</module-name><access-operations>*</access-operations><action>permit</action></rule></rule-list></nacm><routing xmlns="urn:ietf:params:xml:ns:yang:ietf-routing"><routing-instance><name>default</name><description>default-vrf [read-only]</description><routing-protocols><routing-protocol><type>static</type><name>1</name></routing-protocol></routing-protocols></routing-instance></routing></data></rpc-reply>



USE PYTHON

Prettify xml

```python
from ncclient import manager
import xml.dom.minidom as p

m = manager.connect(
    host="192.168.56.101",
    port=830,
    username="cisco",
    password="cisco123!",
    hostkey_verify=False
    )

netconf_reply = m.get_config(source="running")
print(p.parseString(netconf_reply.xml).toprettyxml())
```

```
.........
<routing-protocols>
                        <routing-protocol>
                            <type>static</type>
                            <name>1</name>
                        </routing-protocol>
                    </routing-protocols>
                </routing-instance>
            </routing>
        </data>
</rpc-reply>
```

USE PYTHON

```python
from ncclient import manager
import xml.dom.minidom as p

m = manager.connect(
    host="192.168.56.101",
    port=830,
    username="cisco",
    password="cisco123!",
    hostkey_verify=False
    )

netconf_filter = """
<filter>
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native" />
</filter>
"""
print('#'*80)
netconf_reply = m.get_config(source="running", filter=netconf_filter)
print(p.parseString(netconf_reply.xml).toprettyxml())
```

filter IOS-XE only

```
<diagnostic xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-diagnostics">
                    <bootup>
                        <level>minimal</level>
                    </bootup>
                </diagnostic>
            </native>
        </data>
</rpc-reply>
```

USE PYTHON

```python
from ncclient import manager
import xml.dom.minidom as p

m = manager.connect(
    host="192.168.56.101",
    port=830,
    username="cisco",
    password="cisco123!",
    hostkey_verify=False
    )

netconf_hostname = """
<config>
  <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
    <hostname>R1</hostname>
  </native>
</config>
"""

print('#'*80)
netconf_reply = m.edit_config(target="running", config=netconf_hostname)
print(p.parseString(netconf_reply.xml).toprettyxml())
```

Config Hostname

edit config

```
################################################################################
##
<?xml version="1.0" ?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
message-id="urn:uuid:7861a18b-10be-4051-bdc4-99a0741f1754">
    <ok/>
</rpc-reply>
```

good

USE PYTHON

lo ϕ

```
    hostkey_verify=False
    )

netconf_loopback = """
<config>
 <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
  <interface>
   <Loopback>
    <name>0</name>
    <description>I Created This Loopback via NETCONF</description>
    <ip>
     <address>
      <primary>
       <address>1.1.1.1</address>
       <mask>255.255.255.0</mask>
      </primary>
     </address>
    </ip>
   </Loopback>
  </interface>
 </native>
</config>
"""

print('#'*80)
netconf_reply = m.edit_config(target="running", config=netconf_loopback)
print(p.parseString(netconf_reply.xml).toprettyxml())
```

Config Loopback ϕ IP-addr

```
################################################################################
<?xml version="1.0" ?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
message-id="urn:uuid:03bb1f9a-5226-4e89-8228-8759b5db8f95">
        <ok/>
</rpc-reply>
```

good

USE PYTHON

```python
from ncclient import manager
import xml.dom.minidom as p

m = manager.connect(
    host="192.168.56.101",
    port=830,
    username="cisco",
    password="cisco123!",
    hostkey_verify=False
    )

netconf_filter = """
<filter>
  <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>Loopback0</name>
    </interface>
  </interfaces>
  <interfaces-state xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>Loopback0</name>
    </interface>
  </interfaces-state>
</filter>
"""
print('#'*80)
netconf_reply = m.get(netconf_filter)
#netconf_reply = m.get(netconf_filter)
print(p.parseString(netconf_reply.xml).toprettyxml())
```

int info

int status

get info from device, not from running

```
                        <out-multicast-pkts>0</out-multicast-pkts>
                    <out-discards>0</out-discards>
                    <out-errors>0</out-errors>
                </statistics>
            </interface>
        </interfaces-state>
    </data>
</rpc-reply>
```

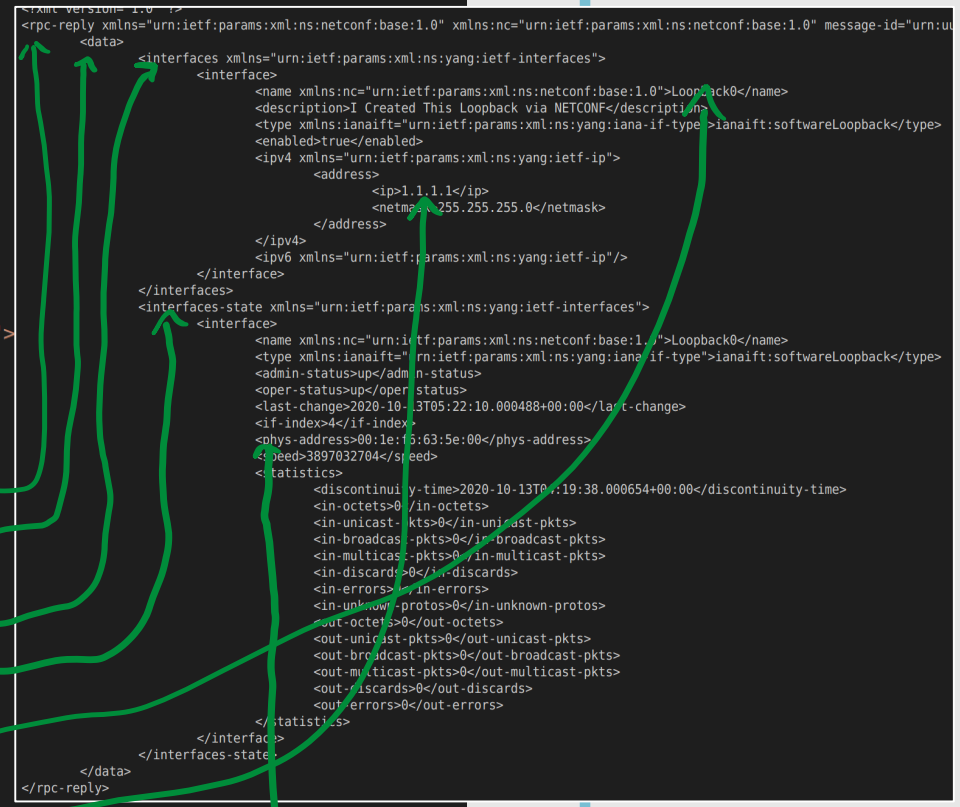
full output of last slide

```
<?xml version="1.0" ?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="urn:uu
  <data>
    <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
      <interface>
        <name xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">Loopback0</name>
        <description>I Created This Loopback via NETCONF</description>
        <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift:softwareLoopback</type>
        <enabled>true</enabled>
        <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
          <address>
            <ip>1.1.1.1</ip>
            <netmask>255.255.255.0</netmask>
          </address>
        </ipv4>
        <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"/>
      </interface>
    </interfaces>
    <interfaces-state xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
      <interface>
        <name xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">Loopback0</name>
        <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">ianaift:softwareLoopback</type>
        <admin-status>up</admin-status>
        <oper-status>up</oper-status>
        <last-change>2020-10-13T05:22:10.000488+00:00</last-change>
        <if-index>4</if-index>
        <phys-address>00:1e:f6:63:5e:00</phys-address>
        <speed>3897032704</speed>
        <statistics>
          <discontinuity-time>2020-10-13T04:19:38.000654+00:00</discontinuity-time>
          <in-octets>0</in-octets>
          <in-unicast-pkts>0</in-unicast-pkts>
          <in-broadcast-pkts>0</in-broadcast-pkts>
          <in-multicast-pkts>0</in-multicast-pkts>
          <in-discards>0</in-discards>
          <in-errors>0</in-errors>
          <in-unknown-protos>0</in-unknown-protos>
          <out-octets>0</out-octets>
          <out-unicast-pkts>0</out-unicast-pkts>
          <out-broadcast-pkts>0</out-broadcast-pkts>
          <out-multicast-pkts>0</out-multicast-pkts>
          <out-discards>0</out-discards>
          <out-errors>0</out-errors>
        </statistics>
      </interface>
    </interfaces-state>
  </data>
</rpc-reply>
```

USE PYTHON

```python
from ncclient import manager
import xml.dom.minidom as p
import xmltodict

m = manager.connect(
    host="192.168.56.101",
    port=830,
    username="cisco",
    password="cisco123!",
    hostkey_verify=False
    )

netconf_filter = """
<filter>
  <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>Loopback0</name>
    </interface>
  </interfaces>
  <interfaces-state xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>Loopback0</name>
    </interface>
  </interfaces-state>
</filter>
"""
print('#'*80)
netconf_reply = m.get(netconf_filter)
intf_details = xmltodict.parse(netconf_reply.xml)["rpc-reply"]["data"]
intf_config = intf_details["interfaces"]["interface"]
intf_info = intf_details["interfaces-state"]["interface"]

print("")
print("Interface Details:")
print("  Name: {}".format(intf_config["name"]["#text"]))
print("  Description: {}".format(intf_config["description"]))
print("  IPv4: {}".format(intf_config["ipv4"]["address"]["ip"]))
print("  Type: {}".format(intf_config["type"]["#text"]))
print("  MAC Address: {}".format(intf_info["phys-address"]))
print("  Packets Input: {}".format(intf_info["statistics"]["in-unicast-pkts"]))
print("  Packets Output: {}".format(intf_info["statistics"]["out-unicast-pkts"]))
```

xml to dic



```
##################################################################

Interface Details:
  Name: Loopback0
  Description: I Created This Loopback via NETCONF
  IPv4: 1.1.1.1
  Type: ianaift:softwareLoopback
  MAC Address: 00:1e:f6:63:5e:00
  Packets Input: 0
  Packets Output: 0
```

We can extract specific information without manually parsing the results.

```
<config>
 <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
  <interface>
   <{int_name}>
    <name>{int_num}</name>
    <description>{int_desc}</description>
    <ip>
     <address>
      <primary>
       <address>{ip_address}</address>
       <mask>{subnet_mask}</mask>
      </primary>
     </address>
    </ip>
   </Loopback>
  </interface>
 </native>
</config>
```

Save to

Config_templ_ietf_interface.xml

```
        hostkey_verify=False
        )

netconf_loopback = """
<config>
 <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
  <interface>
   <Loopback>
    <name>0</name>
    <description>I Created This Loopback via NETCONF</description>
    <ip>
     <address>
      <primary>
       <address>1.1.1.1</address>
       <mask>255.255.255.0</mask>
      </primary>
     </address>
    </ip>
   </Loopback>
  </interface>
 </native>
</config>
"""

print('#'*80)
netconf_reply = m.edit_config(target="running", config=netconf_loopback)
print(p.parseString(netconf_reply.xml).toprettyxml())
```

Slide 24

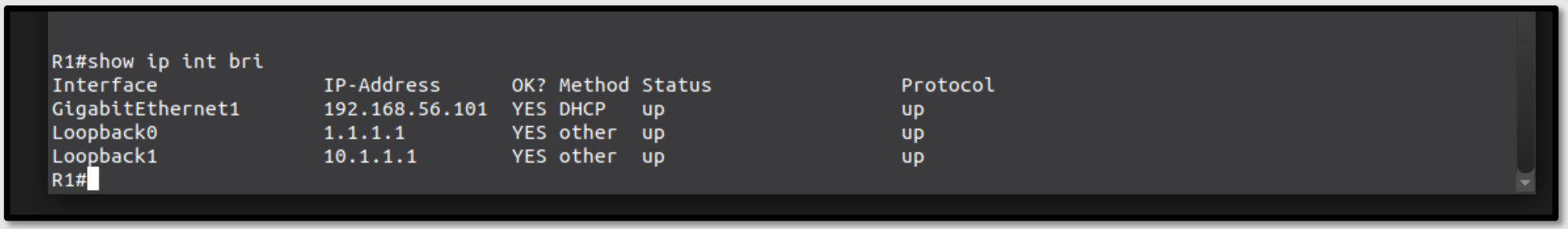
```
router = {"host":"192.168.56.101",
          "port":"830",
          "username":"cisco",
          "password":"cisco123!",
          "hostkey_verify":"False"
        }
```

Save to routers.py

```python
from ncclient import manager
import xml.dom.minidom as p
import xmltodict
import routers

m = manager.connect(**routers.router)

# NETCONF Config Template to use
netconf_template = open("netconf/config_templ_ietf_interface.xml").read()

# Build the XML Configuration to Send
netconf_payload = netconf_template.format(int_name="Loopback",
                                          int_num="1",
                                          int_desc="Configured by NETCONF With a Template",
                                          ip_address="1.1.1.1",
                                          subnet_mask="255.255.255.0"
                                          )
print("Configuration Payload:")
print("----------------------")
print(netconf_payload)

print('#'*80)
# Send NETCONF <edit-config>
netconf_reply = m.edit_config(target="running", config=netconf_payload)

# Print the NETCONF Reply
print(netconf_reply)
```

```
router = {"host":"192.168.56.101",
          "port":"830",
          "username":"cisco",
          "password":"cisco123!",
          "hostkey_verify":"False"
        }
```

```
<config>
 <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
  <interface>
   <{int_name}>
    <name>{int_num}</name>
    <description>{int_desc}</description>
    <ip>
     <address>
      <primary>
       <address>{ip_address}</address>
       <mask>{subnet_mask}</mask>
      </primary>
     </address>
    </ip>
   </Loopback>
  </interface>
 </native>
</config>
```

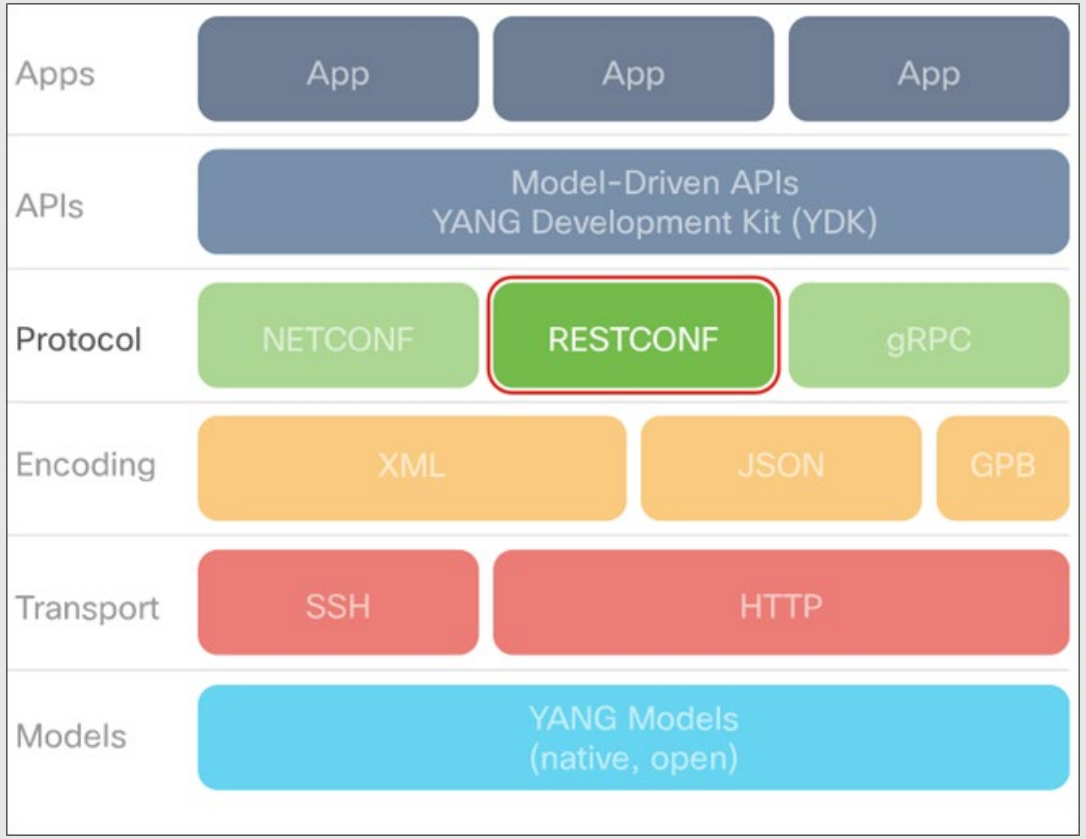

R1#show ip int bri
Interface   IP-Address   OK? Method Status   Protocol
GigabitEthernet1   192.168.56.101 YES DHCP   up   up
Loopback0   1.1.1.1   YES other up   up
Loopback1   10.1.1.1   YES other up   up
R1#

RESTCONF

Agent based / Model-Driven API

What is RESTCONF?

- The RESTCONF RFC 8040 defines a protocol and mechanism for REST-like access to configuration information and control.
- RESTCONF uses datastore models and command verbs defined in NETCONF, encapsulated in HTTP messages.
- RESTCONF uses structured data (XML or JSON) and YANG to provide REST-like APIs, enabling programmatic access to devices.
- RESTCONF is not intended to replace NETCONF, instead provide an HTTP interface that follows the REST principles and is compatible with the NETCONF datastore model.



What is RESTCONF? (Contd.)

RESTCONF vs. NETCONF

Overall, NETCONF is more comprehensive, flexible, and complex than RESTCONF. The differences between NETCONF and RESTCONF:

- RESTCONF is easier to learn and use for engineers with previous REST API experience.
- NETCONF supports running and candidate data stores, while RESTCONF supports only a running data store as any edits of candidate data store are immediately committed.
- RESTCONF does not support obtaining or releasing a data store lock. If a data store has an active lock, the RESTCONF edit operation will fail.
- A RESTCONF edit is a transaction limited to a single RESTCONF call.
- RESTCONF does not support transactions across multiple devices.
- Validation is implicit in every RESTCONF editing operation, which either succeeds or fails.

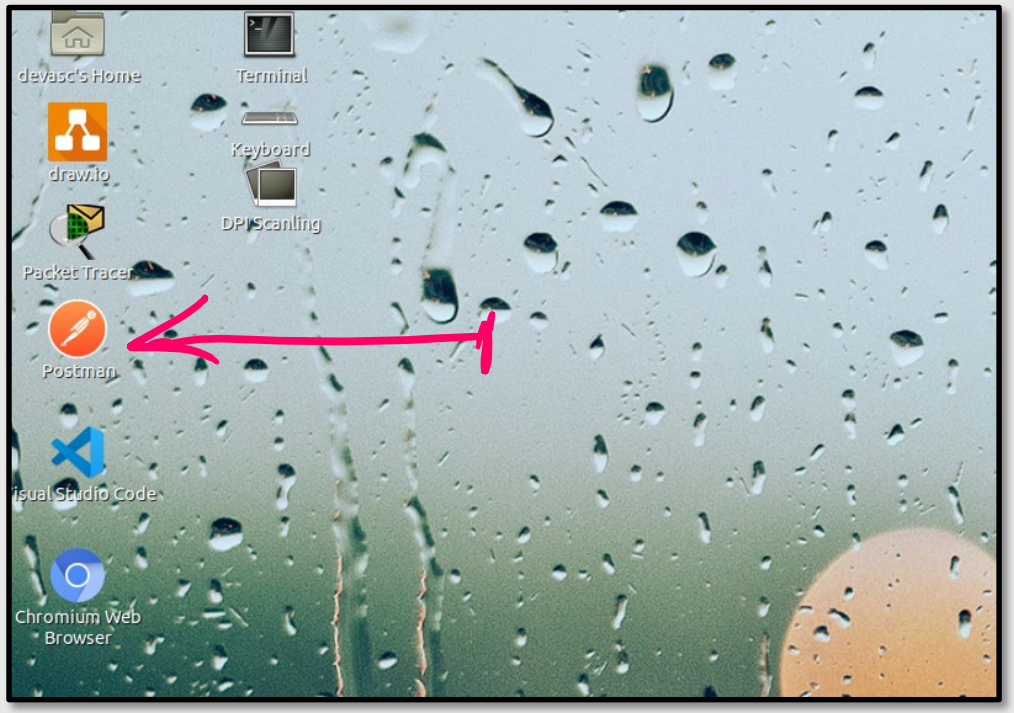
What is RESTCONF? (Contd.)

Description	NETCONF	RESTCONF
Create a data resource	<edit-config>, </edit-config>	POST
Retrieve data and metadata	<get-config>, <get> , </get-config>	GET
Create or replace a data resource	<edit-config> (nc:operation="create/replace")	PUT
Delete a data resource	<edit-config> (nc:operation="delete")	DELETE

- The RESTCONF RFC 8040 states that RESTCONF base URI syntax is /restconf/<resource-type>/<yang-module:resource>. <resource-type> and <yang-module:resource> are variables and the values are obtained using specific YANG model files.
- The basic format of a RESTCONF URL is https://<hostURL>/restconf<resource><container><leaf><options> where any portion after restconf could be omitted.

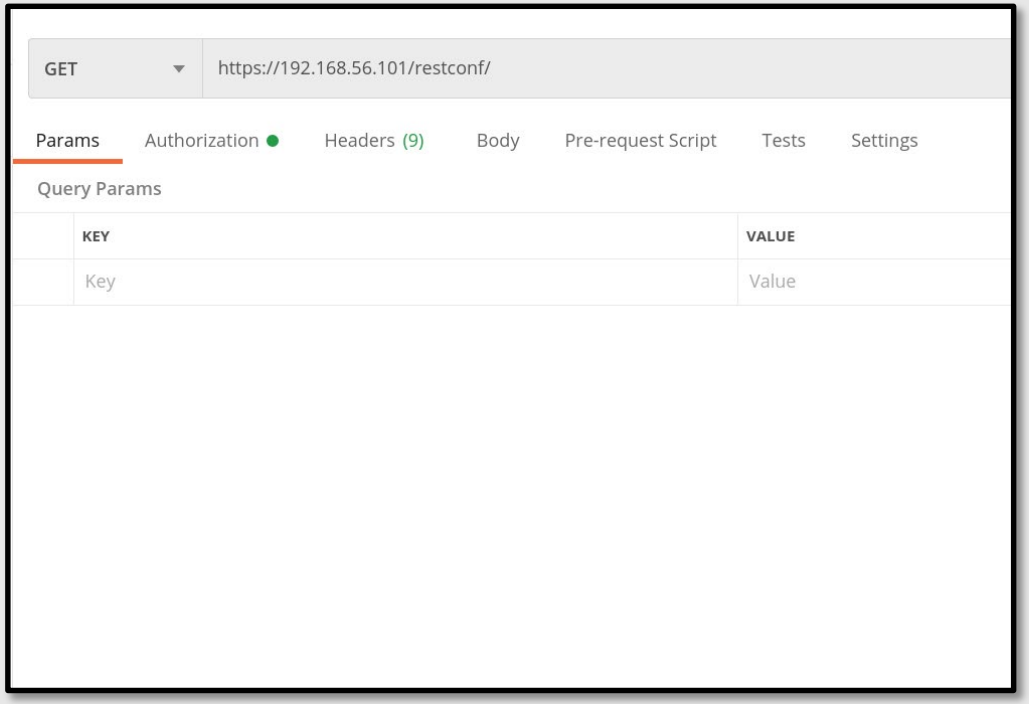
Playing with Postman

- Postman is a software development tool. It enables people to test calls to APIs. Postman users enter data. The data is sent to a special web server address. Typically, information is returned, which Postman presents to the user.
- You will use Postman to send a GET request to the CSR1kv to verify that you can connect to the RESTCONF service. Simply double click the Postman icon on the desktop of your LABVM.



Playing with Postman (Contd.)

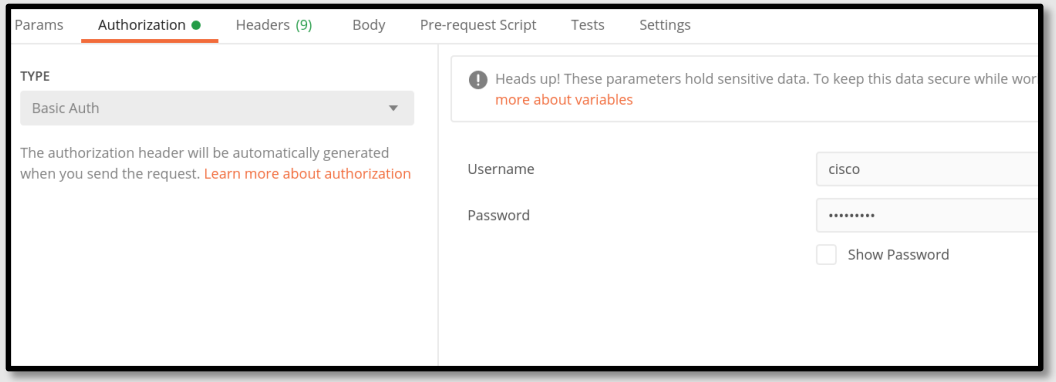
1. In the center, you will see the Launchpad. You can explore this area if you wish.
2. Click the plus sign (+) next to the Launchpad tab to open a GET Untitled Request. This interface is where you will do all of your work in this lab.
3. The request type is already set to GET. Leave the request type set to GET.
4. In the "Enter request URL" field, type in the URL that will be used to access the RESTCONF service that is running on the CSR1kv: *https://192.168.56.101/restconf/*

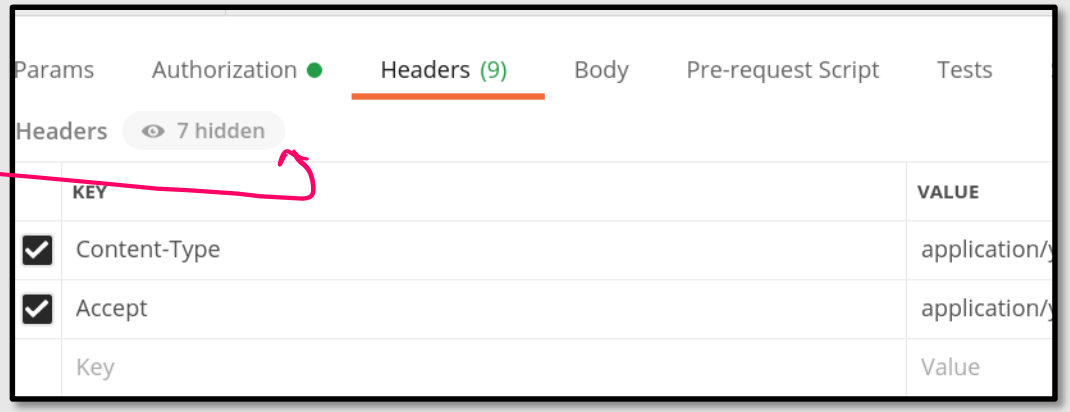


Playing with Postman (Contd.)

Under the URL field, there are tabs listed for Params, Authorization, Headers, Body, Pre-request Script, Test, and Settings. In this lab, you will use Authorization, Headers, and Body.

1. Click the Authorization tab.
2. Under Type, click the down arrow next to "Inherit auth from parent" and choose Basic Auth.
3. For Username and Password, enter the local authentication credentials for the CSR1kv: Username: cisco, Password: cisco123!
4. Click Headers. Then click the 7 hidden. You can verify that the Authorization key has a Basic value that will be used to authenticate the request when it is sent to the CSR1kv.

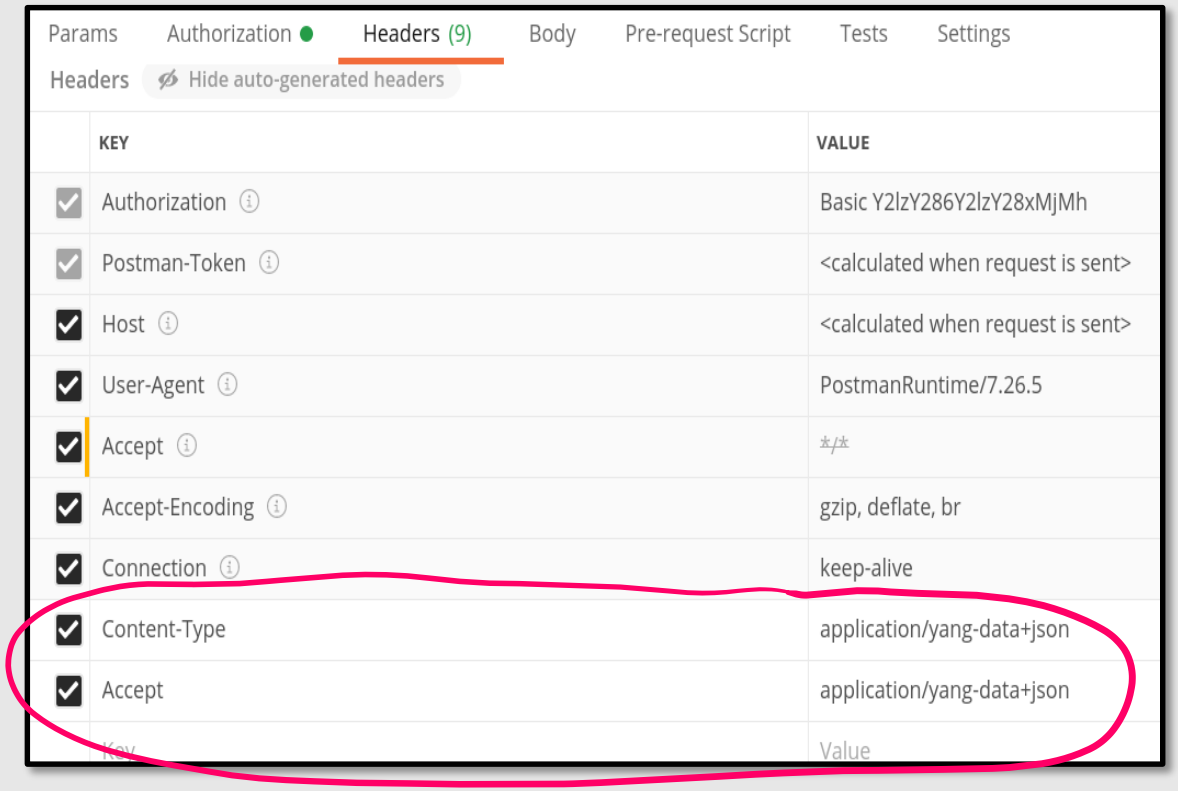




Playing with Postman (Contd.)

You can send and receive data from the CSR1kv in XML or JSON format. For this lab, you will use JSON.

1. In the Headers area, click in the first blank Key field and type Content-Type for the type of key. In the Value field, type application/yang-data+json. This tells Postman to send JSON data to the CSR1kv.
2. Below your Content-Type key, add another key/value pair. The Key field is Accept and the Value field is application/yang-data+json

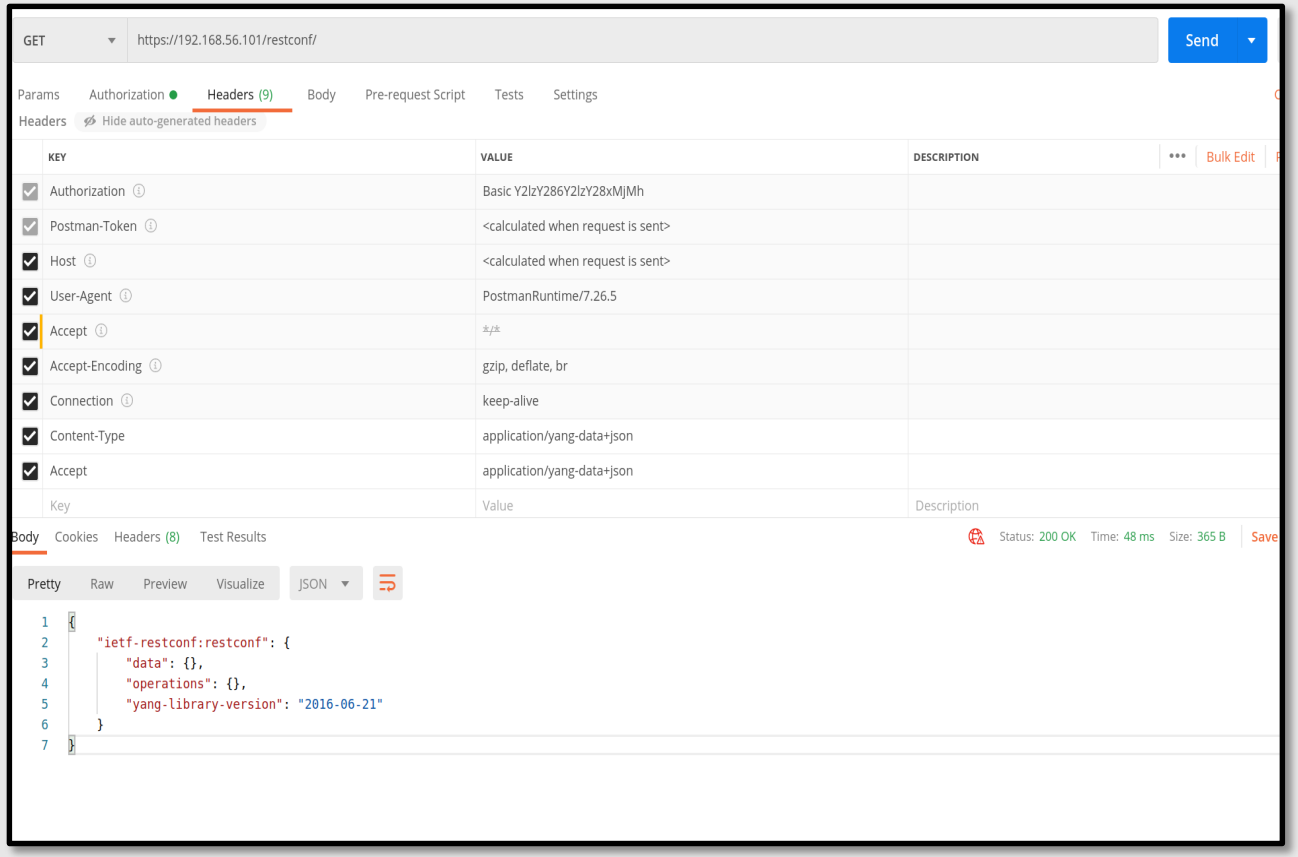


Playing with Postman (Contd.)

Postman now has all the information it needs to send the GET request. Click Send. Below Temporary Headers, you should see the following JSON response from the CSR1kv. If not, verify that you completed the previous steps in this part of the lab and correctly configured RESTCONF and HTTPS service in Part 2.

```
{
        "ietf-restconf:restconf": {
            "data": {},
            "operations": {},
            "yang-library-version": "2016-06-21"
        }
}
```

This JSON response verifies that Postman can now send other REST API requests to the CSR1kv.

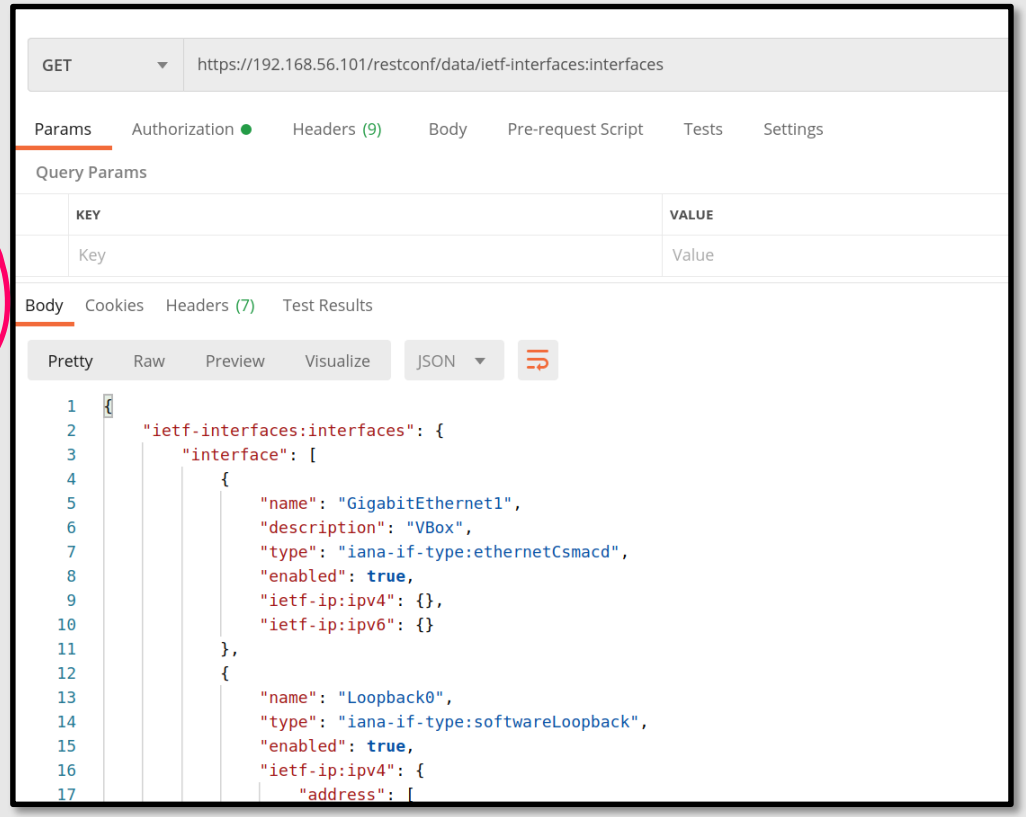


Playing with Postman (Contd.)

1. Now that you have a successful GET request, you can use it as a template for additional requests. At the top of Postman, next to the Launchpad tab, right-click the GET tab that you just used and choose Duplicate Tab.
2. Use the ietf-interfaces YANG model to gather interface information. For the URL, add data/ietf-interfaces:interfaces: https://192.168.56.101/restconf/data/ietf-interfaces:interfaces
3. Click Send. You should see a JSON response from the CSR1kv that is similar to the output shown below. Your output may be different depending on your particular router.

How do I know this URL?

Click here



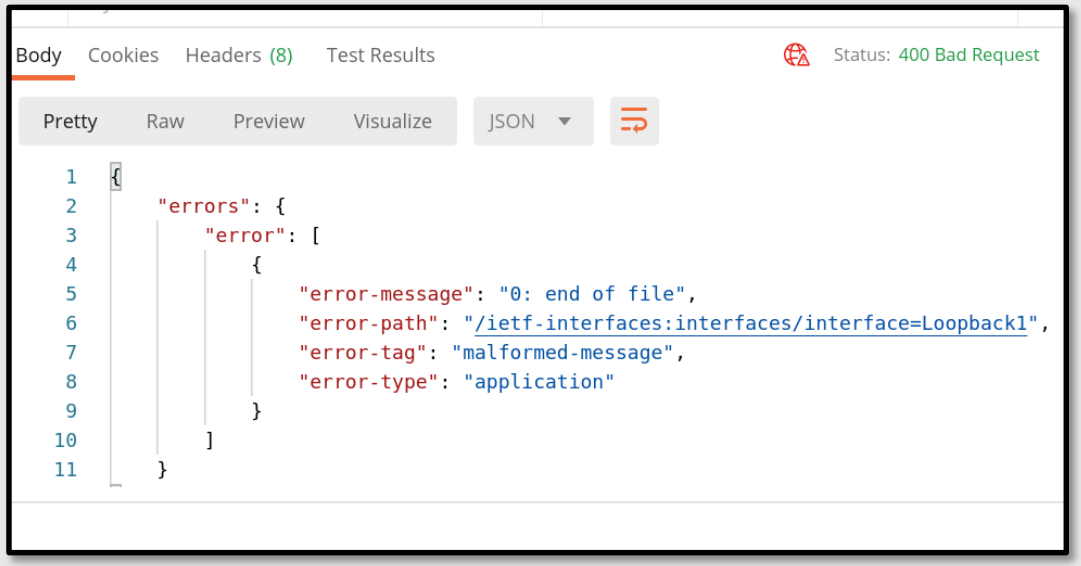

```
{
  "ietf-interfaces:interfaces": {
    "interface": [
      {
        "name": "GigabitEthernet1",
        "description": "VBox",
        "type": "iana-if-type:ethernetCsmacd",
        "enabled": true,
        "ietf-ip:ipv4": {},
        "ietf-ip:ipv6": {}
      }
    ]
  }
}
```

Playing with Postman (Contd.)

In this slide, you will configure Postman to send a **PUT request** to the CSR1kv to create a new loopback interface.

1. Duplicate the last GET request.
2. For the Type of request, click the down arrow next to GET and choose PUT.
3. For the interface= parameter, change it to =Loopback1 to specify a new interface. *https://192.168.56.101/restconf/data/ietf-interfaces:interfaces/interface=Loopback1*

Note: . If you click Send now, you will get error code 400 Bad Request because Loopback1 does not exist yet and you did not provide enough information to create the interface.

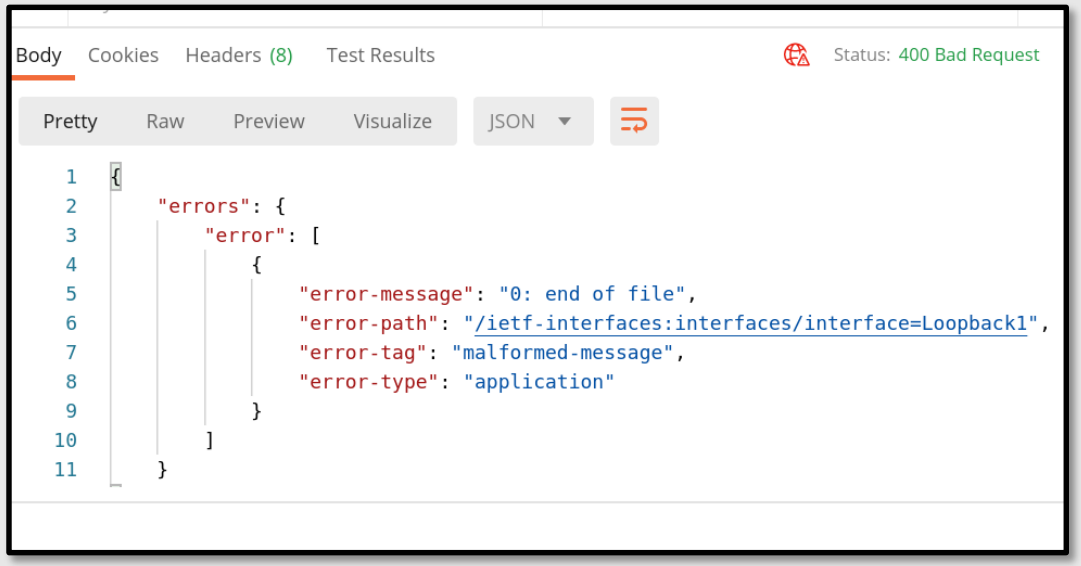


Playing with Postman (Contd.)

In this slide, you will configure Postman to send a **PUT request** to the CSR1kv to create a new loopback interface.

1. Duplicate the last GET request.
2. For the Type of request, click the down arrow next to GET and choose PUT.
3. For the interface= parameter, change it to =Loopback1 to specify a new interface. *https://192.168.56.101/restconf/data/ietf-interfaces:interfaces/interface=Loopback1*

Note: . If you click Send now, you will get error code 400 Bad Request because Loopback1 does not exist yet and you did not provide enough information to create the interface.



Playing with Postman (Contd.)

1. To send a PUT request, you need to provide the information for the body of the request. Next to the Headers tab, click Body. Then click the Raw radio button. The field is currently empty
2. Fill in the Body section with the required JSON data to create a new Loopback1 interface. You can copy the Body section of the previous GET request and modify it. Or you can copy the following into the Body section of your PUT request. Notice that the type of interface must be set *to softwareLoopback*.

```
{
  "ietf-interfaces:interface": {
    "name": "Loopback1",
    "description": "My first RESTCONF loopback",
    "type": "iana-if-type:softwareLoopback",
    "enabled": true,
    "ietf-ip:ipv4": {
      "address": [
        {
          "ip": "10.1.1.1",
          "netmask": "255.255.255.0"
        }
          ]
    },
    "ietf-ip:ipv6": {}
  }
}
```

Playing with Postman (Contd.)

PUT https://192.168.56.101/restconf/data/ietf-interfaces:interfaces/interface=Loopback1

arams Authorization Headers (11) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
  "ietf-interfaces:interface": {
    "name": "Loopback1",
    "description": "My first RESTCONF loopback",
    "type": "iana-if-type:softwareLoopback",
    "enabled": true,
    "ietf-ip:ipv4": {
      "address": [
        {
          "ip": "10.1.1.1",
          "netmask": "255.255.255.0"
        }
      ]
    },
    "ietf-ip:ipv6": {}
  }
}
```

ody Cookies Headers (10) Test Results Status: 201 Created Time:

Pretty Raw Preview Visualize HTML

1

OK!

```
SR1kv(config)#do show ip int brie
nterface                  IP-Address      OK? Method Status                Protocol
igabitEthernet1           192.168.56.101  YES DHCP   up                    up
oopback1                  10.1.1.1        YES other  up                    up
SR1kv(config)#
```

RESTCONF WITH PYTHON

In next several slides we will talk about one python script:

restconf_3.py

This code provides a collection of functions that are useful for user to monitor CPU and Memory

```python
import json
import requests
import sys
from argparse import ArgumentParser
from collections import OrderedDict
import urllib3
```

Libraries

```python
# Disable SSL Warnings
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

# These variables target the RESTCONF Always-On Sandbox hosted by Cisco DevNet
HOST = '192.168.56.101'
PORT = '443'
USER = 'cisco'
PASS = 'cisco123!'

# Identifies the interface on the device used for management access
# Used to ensure the script isn't used to update the IP leveraged to manage device
MANAGEMENT_INTERFACE = "GigabitEthernet1"

# Create the base URL for RESTCONF calls
#url_base = "https://{h}:{p}/restconf".format(h=HOST, p=PORT)
url_base = "https://{h}/restconf".format(h=HOST)

# Identify yang+json as the data formats
headers = {'Content-Type': 'application/yang-data+json',
           'Accept': 'application/yang-data+json'}
```

router infor

header

https://192.168.56.101/restconf

Base URL

Container

Filename

Auth

1st level

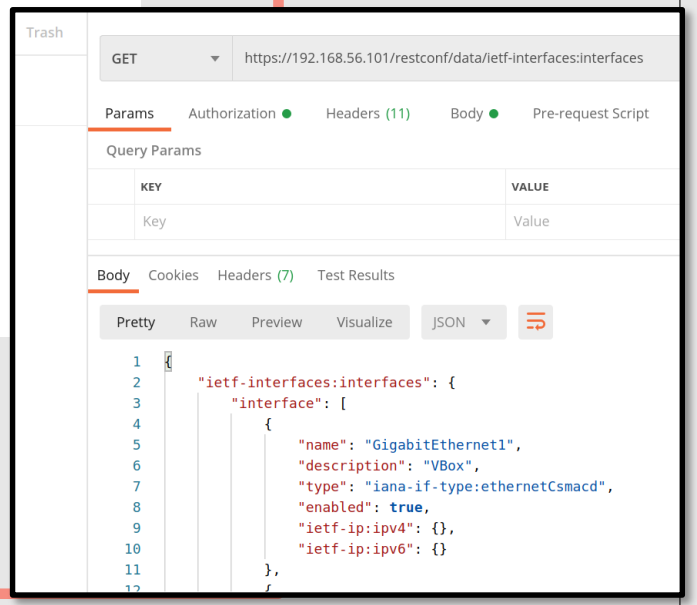
Test ← The function

```
# Function to retrieve the list of interfaces on a device
def get_configured_interfaces():
    url = url_base + "/data/ietf-interfaces:interfaces"

    # this statement performs a GET on the specified url
    response = requests.get(url,
                            auth=(USER, PASS),
                            headers=headers,
                            verify=False
                            )

    # return the json as text
    print(url)
    return response.json()["ietf-interfaces:interfaces"]["interface"]

####### Test Function ########
get_configured_interfaces()
```

Python

```
https://192.168.56.101/restconf/data/ietf-interfaces:interfaces

[{'name': 'GigabitEthernet1',
  'description': 'VBox',
  'type': 'iana-if-type:ethernetCsmacd',
  'enabled': True,
  'ietf-ip:ipv4': {},
  'ietf-ip:ipv6': {}},
 {'name': 'Loopback0',
  'type': 'iana-if-type:softwareLoopback',
  'enabled': True,
  'ietf-ip:ipv4': {'address': [{'ip': '1.1.1.10',
     'netmask': '255.255.255.0'}]},
  'ietf-ip:ipv6': {}}]
```

Postman



function to shutdown an int

```python
def shutdown_int(interface):
    # RESTCONF URL for specific interface
    url = url_base + "/data/ietf-interfaces:interfaces/interface={i}".format(i=interface)

    type = 'iana-if-type:ethernetCsmacd'
    if 'Loopback' in interface:
        type = 'iana-if-type:softwareLoopback'

    # Create the data payload to disbale the interface
    # Need to use OrderedDicts to maintain the order of elements
    data = OrderedDict([('ietf-interfaces:interface',
             OrderedDict([
                          ('name', interface),
                          ('type', type),
                          ('enabled', False) #shutdown
                        ])
                       )])

    # Use PUT request to update data
    response = requests.put(url,
                            auth=(USER, PASS),
                            headers=headers,
                            verify=False,
                            json=data
                            )

    print(url)
    print(response.text)

####### Test Function ########
#shutdown_int('Loopback0')
```

Shut

```
"ietf-interfaces:interfaces": {
    "interface": [
        {
            "name": "GigabitEthernet1",
            "description": "VBox",
            "type": "iana-if-type:ethernetCsmacd",
            "enabled": true,
```

no Shut

UI code → draw a button

```
# some handy functions to use along widgets
from IPython.display import display, Markdown, clear_output
# widget packages
import ipywidgets as widgets

# defining some widgets
button = widgets.Button(description="Emergency Shutdown")
output = widgets.Output()
display(button, output)

# Button callback (action)
def on_button_clicked(b):
    with output:
        print("Emergency Shutdown")
        shutdown_int('Loopback0')

button.on_click(on_button_clicked)
```

→ click action

Emergency Shutdown

← Button

get CPU (5s)

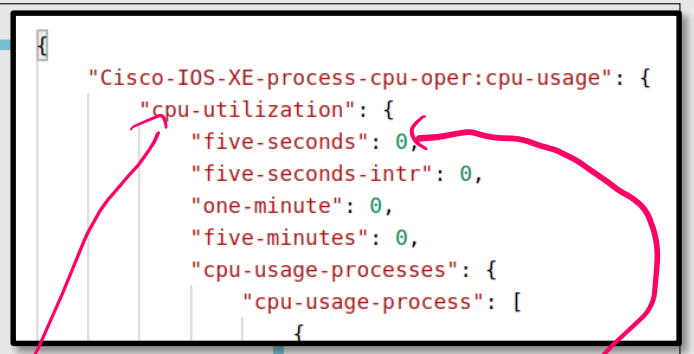
filename

container

```
# Function to retrieve the CPU on a device
def get_cpu():
    url = url_base + "/data/Cisco-IOS-XE-process-cpu-oper:cpu-usage/cpu-utilization/five-seconds"

    # this statement performs a GET on the specified url
    response = requests.get(url,
                            auth=(USER, PASS),
                            headers=headers,
                            verify=False
                            )

    # return the json as text
    #print(url)
    return response.json()['Cisco-IOS-XE-process-cpu-oper:five-seconds']

######## Test Function ########
get_cpu()
```

1st level

Test function

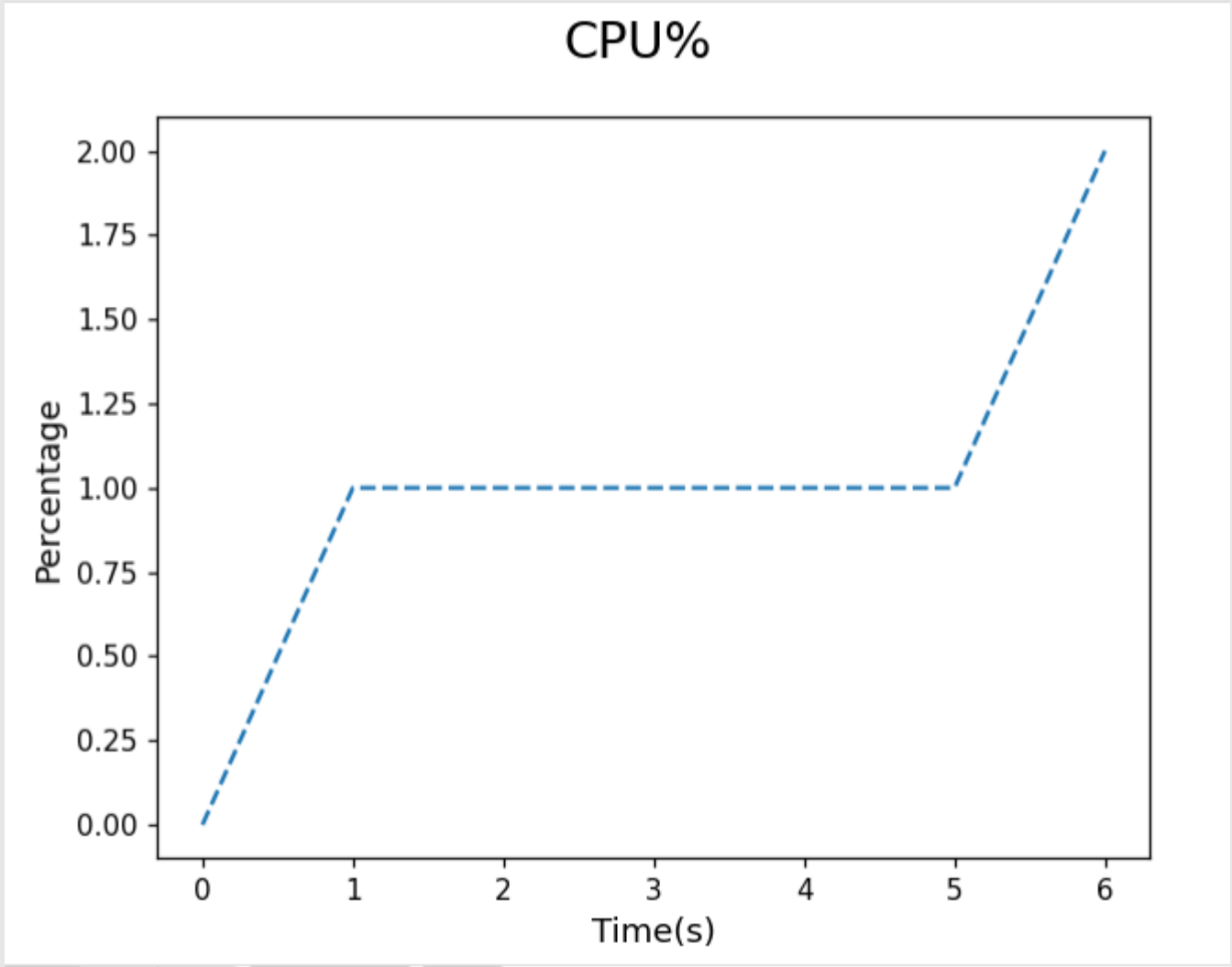


UI Thread

draw Line chart

```
%matplotlib notebook

from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation
from matplotlib import style
from threading import Thread
import time

class LiveLine:
    def __init__(self):
        self.x_data, self.y_data = [], []
        self.figure = plt.figure()
        self.figure.suptitle('CPU%', fontsize=18)
        self.line, = plt.plot(self.x_data, self.y_data,'--')
        plt.xlabel('Time(s)', fontsize=12)
        plt.ylabel('Percentage', fontsize=12)
        self.animation = FuncAnimation(self.figure, self.update, interval=1000)
        self.th = Thread(target=self.thread_f, daemon=True)
        self.th.start()

    def update(self, frame):
        self.line.set_data(self.x_data, self.y_data)
        self.figure.gca().relim()
        self.figure.gca().autoscale_view()
        return self.line,

    def show(self):
        plt.show()

    def thread_f(self):
        x = 0
        while True:
            self.x_data.append(x)
            x += 1
            self.y_data.append(get_cpu())
            time.sleep(1)

g = LiveLine()
g.show()
```

update cpu usage every 1 s


CPU%
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Percentage
0
1
2
3
4
5
6
Time(s)

filename

container

```python
# Function to retrieve memory usase on a device
def get_mem():
    url = url_base + "/data/Cisco-IOS-XE-memory-oper:memory-statistics/memory-statistic=Processor"

    # this statement performs a GET on the specified url
    response = requests.get(url,
                            auth=(USER, PASS),
                            headers=headers,
                            verify=False
                            )
    data = response.json()["Cisco-IOS-XE-memory-oper:memory-statistic"]
    # return the json as text
    print("Name: ", data["name"])
    used,free = 0,0
    try:
        used = int(data["used-memory"])
        free = int(data["free-memory"])
    except KeyError:
        print("Json Error")
    print()

    # return the json as text
    return (used/(used+free)) * 100, (free/(used+free)) * 100

####### Test Function ########
get_mem()
```

```
{
    "Cisco-IOS-XE-memory-oper:memory-statistics": {
        "memory-statistic": [
            {
                "name": "Processor",
                "total-memory": "2234564224",
                "used-memory": "324378828",
                "free-memory": "1910185396",
                "lowest-usage": "1909886608",
                "highest-usage": "1269342172"
```

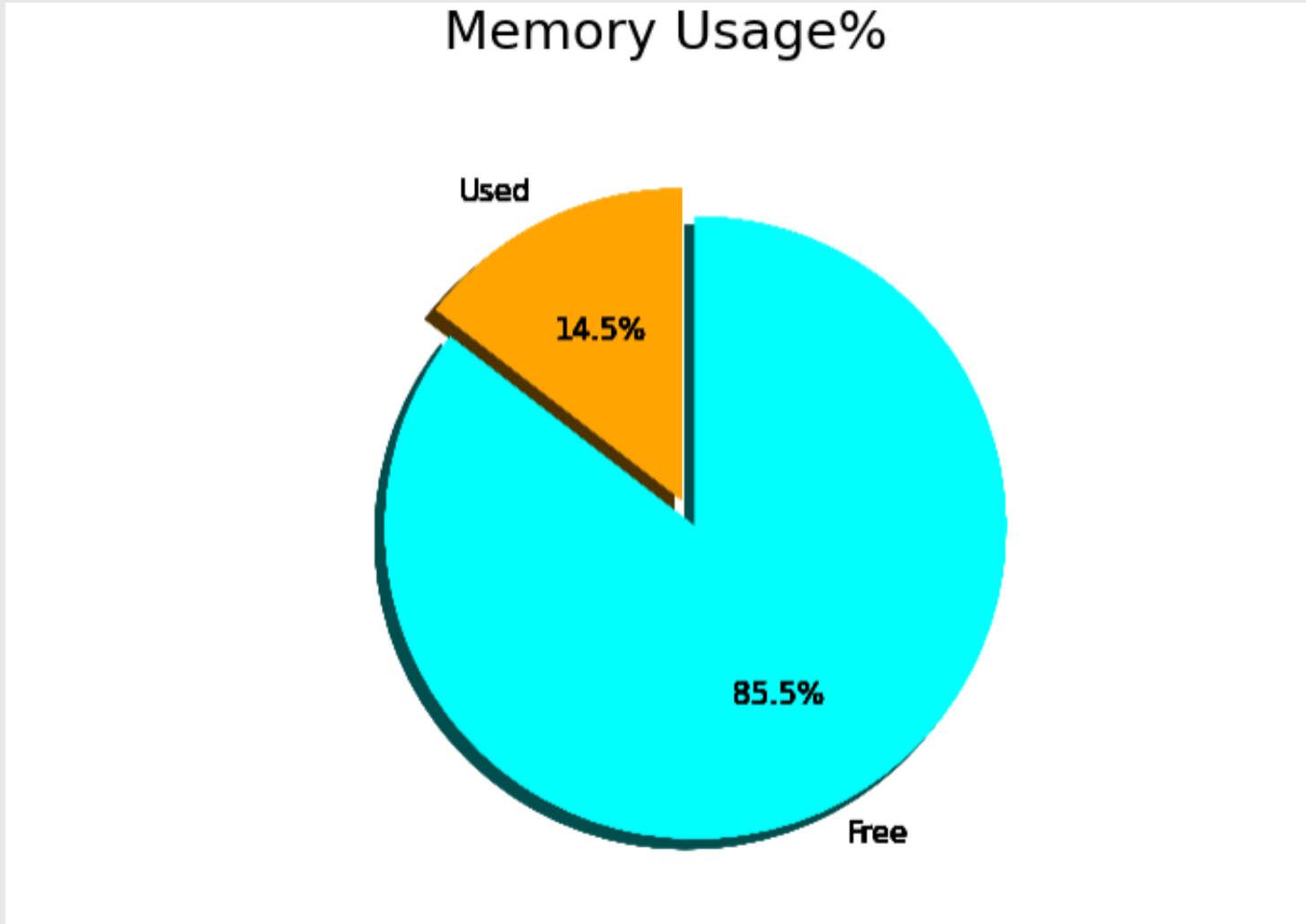
test call

get %

UI thread draw a pie chart

```
class LivePie:
    def __init__(self):
        self.data = [30,70]
        self.figure, self.pie_chart = plt.subplots()
        self.figure.suptitle('Memory Usage%', fontsize=18)
        self.labels = 'Used', 'Free'
        self.explode = (0, 0.1) # only "explode" the 2nd slice
        self.colors = ("orange", "Cyan")
        self.pie_chart.axis('equal')
        self.animation = FuncAnimation(self.figure, self.update, interval=1000)
        self.th = Thread(target=self.thread_f, daemon=True)
        self.th.start()

    def update(self, frame):
        self.pie_chart.pie(self.data, explode=self.explode,labels = self.labels, colors = self.colors,
                           autopct='%1.1f%%',shadow=True, startangle=90)
        self.figure.gca().relim()
        self.figure.gca().autoscale_view()
        return self.pie_chart,

    def show(self):
        print('showing chart!')

    def thread_f(self):
        while True:
            self.data = get_mem()
            plt.pause(0.001)
            plt.draw()
            time.sleep(1)

g = LivePie()
g.show()
```

update memory usage every 1s


Memory Usage%
Used
14.5%
85.5%
Free

END.