Cisco live!

January 29 – February 2, 2018 · Barcelona

BRKDEV-1368

Effectively Understand and Leverage YANG with NETCONF and RESTCONF for Model Driven Programmability

Bryan Byrne, Technical Solutions Architect
ccie 25607, R/S
@bryan25607

Hank Preston, NetDevOps Evangelist
ccie 38336, R/S
@hfpreston

Cisco live!

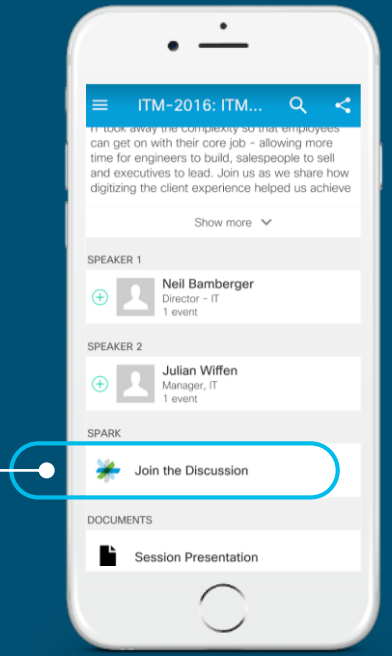
Cisco Spark

Questions?

Use Cisco Spark to communicate with the speaker after the session

How

1. Find this session in the Cisco Live Mobile App
2. Click “Join the Discussion”
3. Install Spark or go directly to the space
4. Enter messages/questions in the space



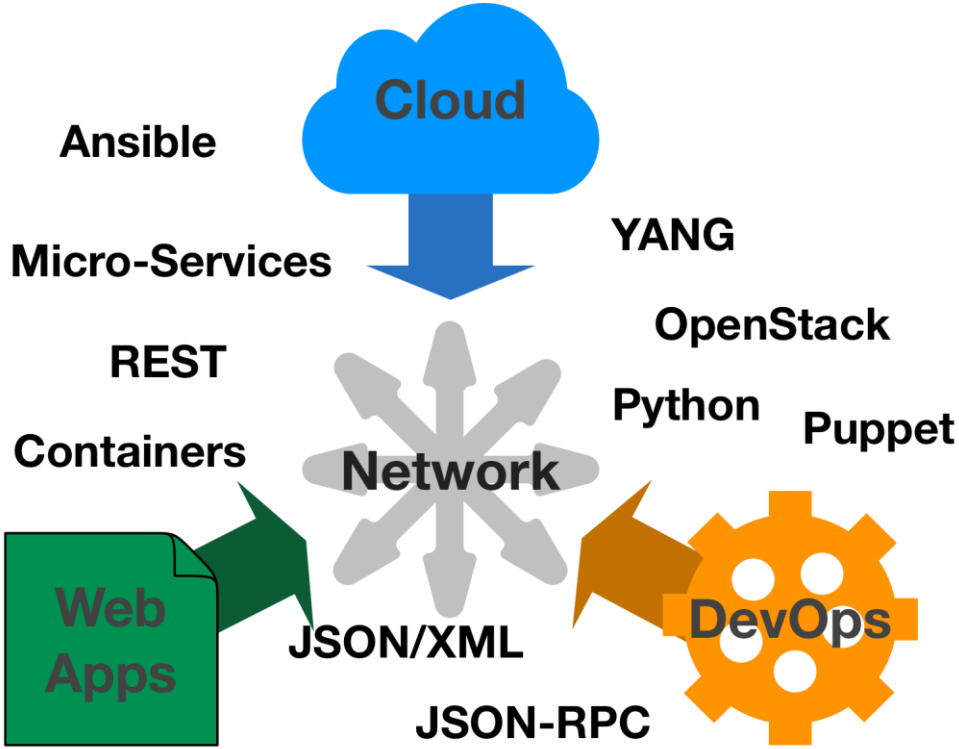
cs.co/ciscolivebot#BRKDEV-1368

Agenda

- The Road to Model Driven Programmability
- Introduction to YANG Data Models
- Introduction to NETCONF
- Introduction to RESTCONF
- Model Driven Programmability in Action
- Model Driven Programmability in Real Life
- Conclusion and Q/A

The Road to Model Driven Programmability

The Network is No Longer Isolated



What about SNMP?

SNMP works "reasonably well for device monitoring"

RFC 3535: Overview of the 2002 IAB Network Management Workshop – 2003
https://tools.ietf.org/html/rfc3535

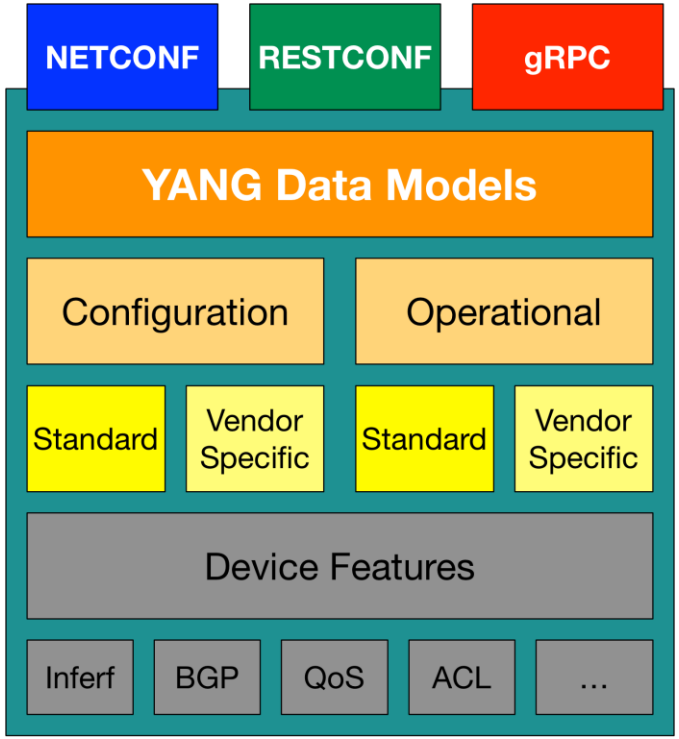
- Typical config: SNMPv2 read-only community strings
- Typical usage: interface statistics queries and traps
- Empirical Observation: SNMP is not used for configuration
  - Lack of Writeable MIBs
  - Security Concerns
  - Difficult to Replay/Rollback
  - Special Applications

RFC 3535: What is Needed?

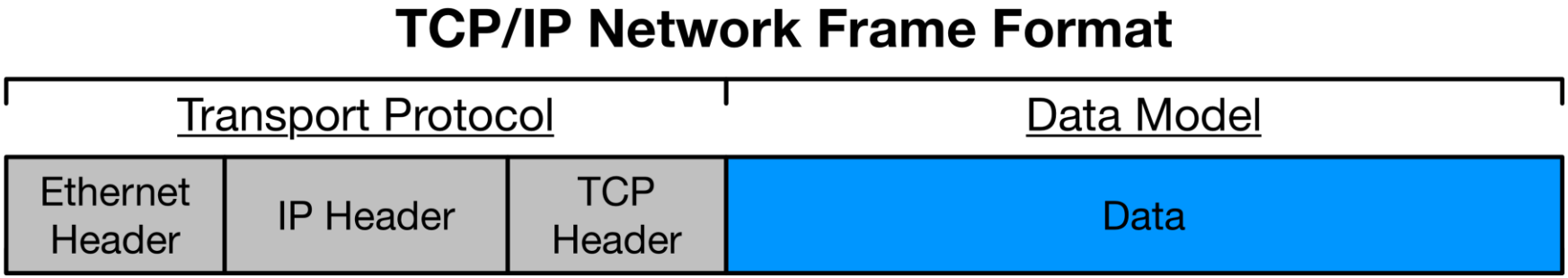
- A programmatic interface for device configuration
- Separation of Configuration and State Data
- Ability to configure "services" NOT "devices"
- Integrated error checking and recovery

Model Driven Programmability

- NETCONF – 2006 – RFC 4741
  (RFC 6241 in 2011)
- YANG – 2010 – RFC 6020
- RESTCONF – 2017 – RFC 8040
- gRPC – 2015 – OpenSource project by Google
  - *Not covered in today's session*



Transport (Protocol) vs Data (Model)

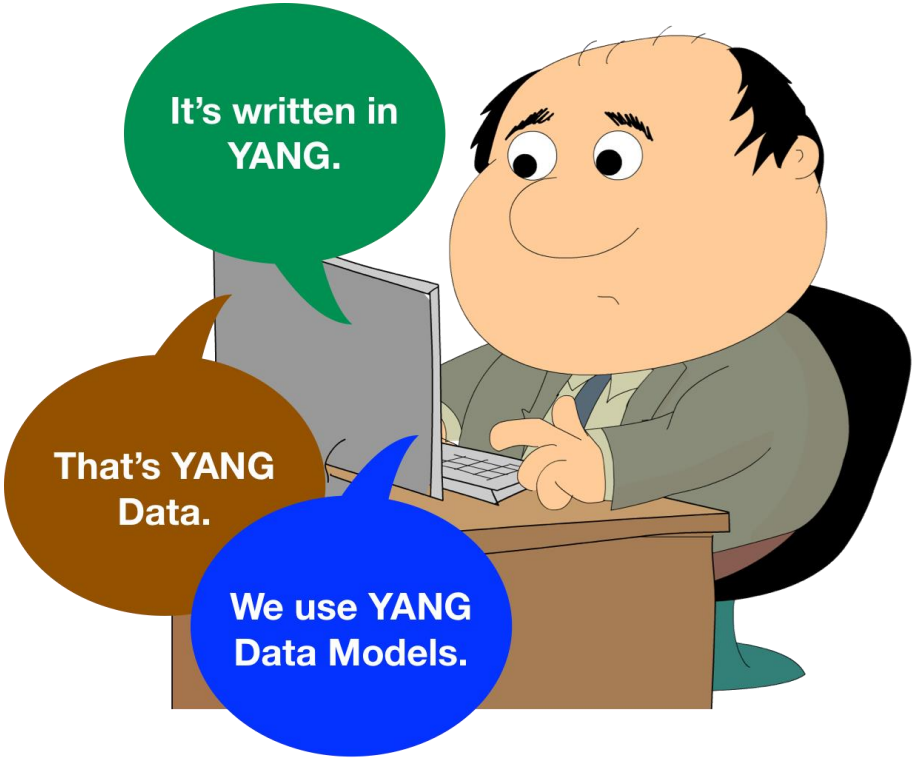


- NETCONF
- RESTCONF
- gRPC

- YANG

What is YANG?

Three Meanings of “YANG”



YANG Modeling Language

- Module that is a self-contained top-level hierarchy of nodes
- Uses containers to group related nodes
- Lists to identify nodes that are stored in sequence
- Each individual attribute of a node is represented by a leaf
- Every leaf must have an associated type

```
module ietf-interfaces {
  import ietf-yang-types {
    prefix yang;
  }
  container interfaces {
    list interface {
      key "name";
      leaf name {
        type string;
      }
      leaf enabled {
        type boolean;
        default "true";
      }
    }
}
```

Example edited for simplicity and brevity

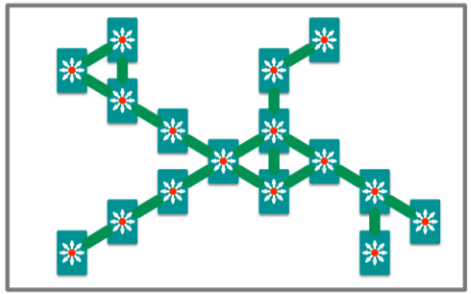
What is a Data Model?

A data model is simply a well understood and agreed upon method to describe "something". As an example, consider this simple "data model" for a person.

- ***Person***
  - **Gender** - male, female, other
  - **Height** - Feet/Inches or Meters
  - **Weight** - Pounds or Kilos
  - **Hair Color** - Brown, Blond, Black, Red, other
  - **Eye Color** - Brown, Blue, Green, Hazel, other

What might a YANG Data Model describe?

Device Data Models
- Interface
- VLAN
- Device ACL
- Tunnel
- OSPF
- etc



Service Data Models
- L3 MPLS VPN
- MP-BGP
- VRF
- Network ACL
- System Management
- Network Faults
- etc

Working with YANG Data Models

Where do Models Come From?

- **Standard definition**
  (IETF, ITU, OpenConfig, etc.)
- Compliant with standard
  `ietf-diffserv-policy.yang`
  `ietf-diffserv-classifer.yang`
  `ietf-diffserv-target.yang`

- **Vendor definition**
  (i.e. Cisco)
- Unique to Vendor Platforms
  `cisco-memory-stats.yang`
  `cisco-flow-monitor`
  `cisco-qos-action-qlimit-cfg`

https://github.com/YangModels/yang

What is OpenConfig?

Models Designed by Operators for Operators

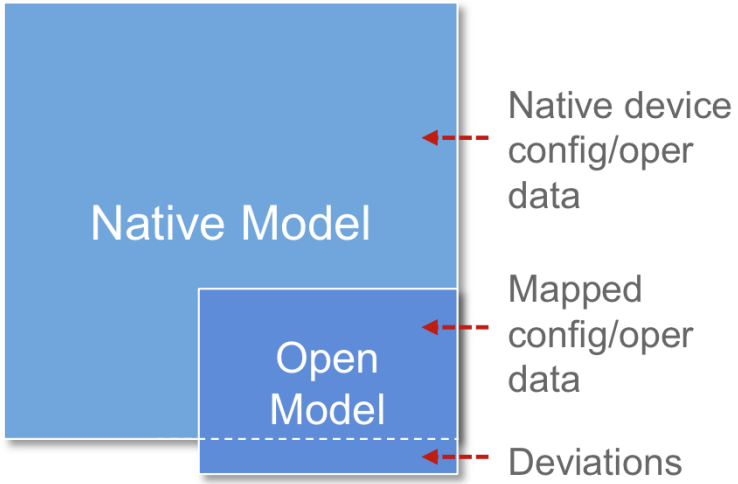
“OpenConfig’s initial focus is on compiling a consistent set of vendor-neutral data models based on actual operational needs from use cases and requirements from multiple network operators.”

OpenConfig FAQ:
www.openconfig.com

What is OpenConfig?

Models Designed by Operators for Operators

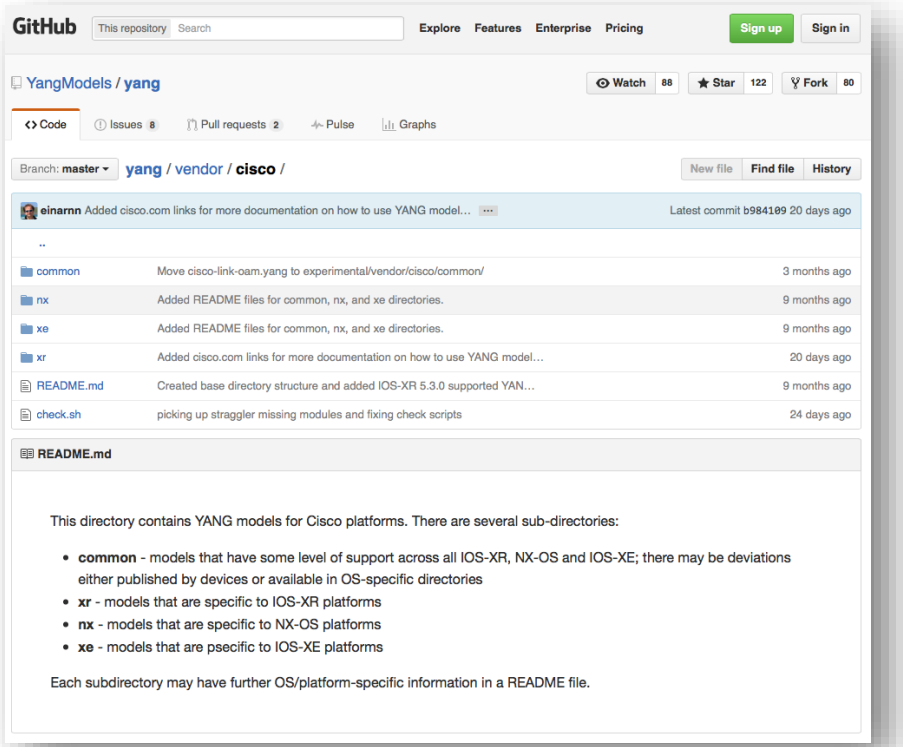
- Focused on creating vendor-neutral data models written in YANG
- Models combine both configuration and operational data
- Model coverage still limited with an active development community
- Support from multiple routing vendors (e.g. Cisco, Juniper, Arista)
  - Vendor exceptions carried a deviations



Where to get the Models?

- For YANG modules from standard organizations such as the IETF, open source such as Open Daylight or vendor specific modules”
  - https://github.com/YangModels/yang

- For OpenConfig models
  - https://github.com/openconfig/public



YANG Data Models

The model can be displayed and represented in any number of formats depending on needs at the time. Some options include:

- YANG Language
- Clear Text
- XML
- JSON
- HTML/JavaScript

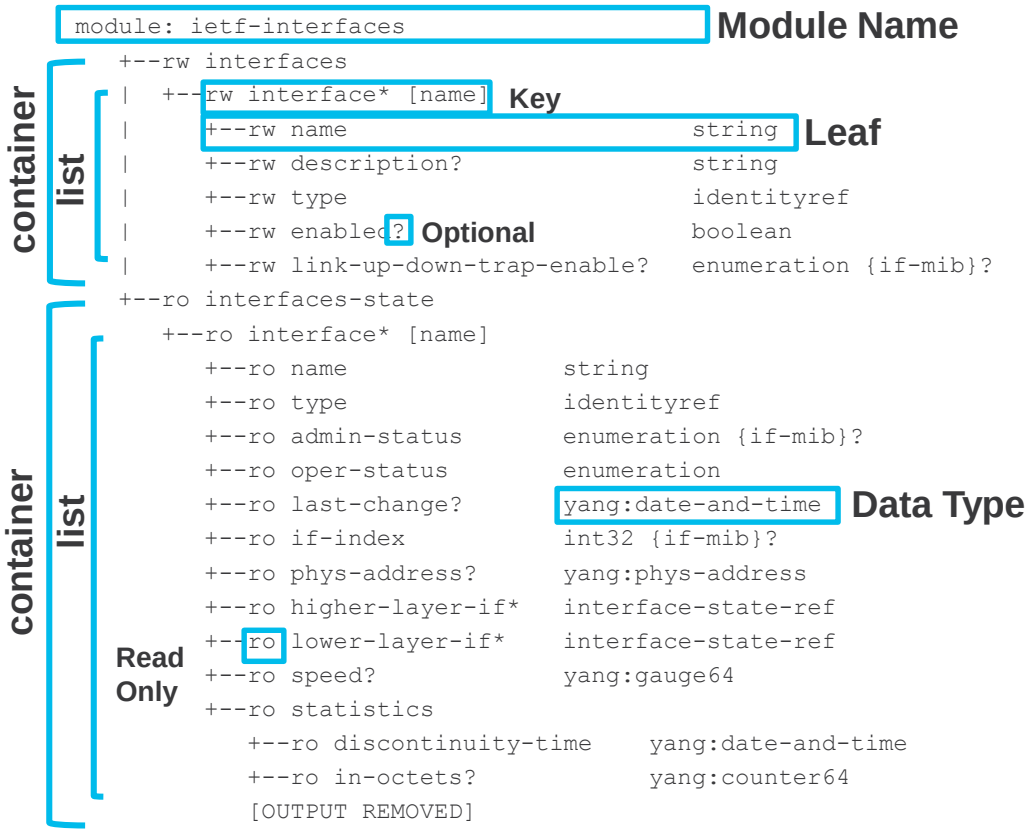
Working with YANG Models

```
DevNet$ pyang -f tree ietf-interfaces.yang

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}?
```

Example output edited for simplicity and brevity

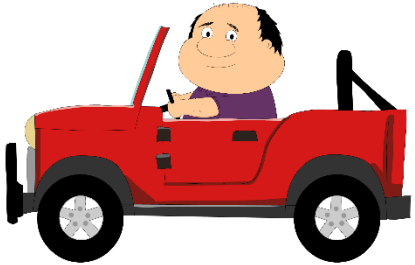
Using pyang

- Python YANG Library
- Validate and display YANG files
- Many formats for display
  - Text: tree
  - HTML: jstree



Example edited for simplicity and brevity

Augmentation and Deviations in YANG



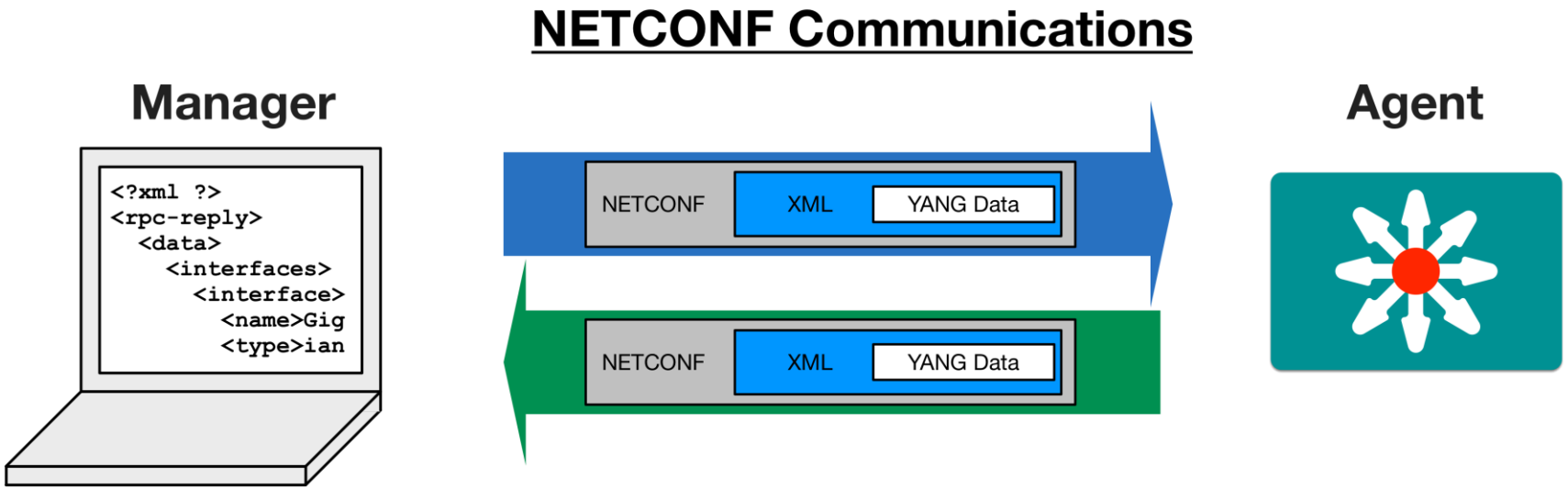
Standard Model

Standard Model with Augmentation

Standard Model with Deviation

Network Device Data in YANG

Actual Device Data Modeled in YANG



Use NETCONF to Retrieve ietf-interfaces data

- NETCONF details covered in another session
- `ncclient` provides a Python client for NETCONF
- Using built-in library to print reply
  - `xml.dom.minidom`

```
from device_info import ios_xe1
from ncclient import manager
import xml.dom.minidom

# NETCONF filter to use
netconf_filter = open("filter-ietf-interfaces.xml").read()


if __name__ == '__main__':
    with manager.connect(host=ios_xe1["address"], port=ios_xe1["port"],
                         username=ios_xe1["username"],
                         password=ios_xe1["password"],
                         hostkey_verify=False) as m:

        netconf_reply = m.get_config("running", netconf_filter)
        interfaces = xml.dom.minidom.parseString(netconf_reply.xml)
        interfaces = interfaces.getElementsByTagName("interfaces")
        print(interfaces[0].toprettyxml())
```

BRKDEV-1368/yang/device_info.py
BRKDEV-1368/yang/example1.py

Use NETCONF to Retrieve ietf-interfaces data

```
DevNet$ python example1.py
```


```
<interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
        <interface>
                <name>GigabitEthernet1</name>
                <description>DON'T TOUCH ME</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">
                        <address>
                                <ip>10.10.10.48</ip>
                                <netmask>255.255.255.0</netmask>
                        </address>
                </ipv4>
                <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"/>
        </interface>
        <interface>
                <name>GigabitEthernet2</name>
                <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>
```

BRKDEV-1368/yang/device_info.py
BRKDEV-1368/yang/example1.py

YANG Model Augmentation

```
<interface>
 <name>GigabitEthernet2</name>
 <description>**THIS IS INTERFACE 2**</description>
 <type>ianaift:ethernetCsmacd</type>
  <enabled>true</enabled>
  <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
    <address>
      <ip>172.16.20.1</ip>
       <netmask>255.255.255.0</netmask>
    </address>
  </ipv4>
</interface>
```

```
(venv) $pyang -f tree ietf-interfaces.yang
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}?
```

Where is the <ipv4> leaf in the model?

YANG Model Augmentation

```
<interface>
 <name>GigabitEthernet2</name>
 <description>**THIS IS INTERFACE 2**</description>
 <type>ianaift:ethernetCsmacd</type>
  <enabled>true</enabled>
  <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
    <address>
      <ip>172.16.20.1</ip>
       <netmask>255.255.255.0</netmask>
    </address>
  </ipv4>
</interface>
```

```
(venv) $pyang -f tree ietf-interfaces.yang
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}?
```

Where is the <ipv4> leaf in the model?

```
module: ietf-ip
  augment /if:interfaces/if:interface:
    +--rw ipv4!
    |  +--rw enabled?      boolean
    |  +--rw forwarding?   boolean
    |  +--rw mtu?          uint16
    |  +--rw address* [ip]
    |  |  +--rw ip               inet:ipv4-address-no-zone
    |  |  +--rw (subnet)
    |  |     +--:(prefix-length)
    |  |     |  +--rw prefix-length?   uint8
    |  |     +--:(netmask)
    |  |        +--rw netmask?         yang:dotted-quad {ipv4-non-contiguous-netmasks}?
```

The YANG model states what base model it is augmenting.

YANG Model Deviations

```
module: ietf-ip
  augment /if:interfaces-state/if:interface:
    +--ro ipv4!
    |  +--ro forwarding?   boolean
    |  +--ro mtu?          uint16
    |  +--ro address* [ip]
    |  |  +--ro ip           inet:ipv4-address-no-zone
    |  |  +--ro (subnet)?
    |  |  |  +--:(prefix-length)
    |  |  |  |  +--ro prefix-length?   uint8
    |  |  |  +--:(netmask)
    |  |  |     +--ro netmask?     yang:dotted-quad
    |  |  +--ro origin?          ip-address-origin
```

Where is the <ipv4> leaf in the device data? ←

```
<interfaces-state xmlns="ietf-interfaces">
 <interface>
  <name>GigabitEthernet1</name>
  <type>ianaift:ethernetCsmacd</type>
  <admin-status>up</admin-status>
  <oper-status>up</oper-status>
  <last-change>2017-11-14T13:33:</last-change>
  <if-index>0</if-index>
  <phys-address>00:50:56:bb:c9:2c</phys-address>
  <speed>1024000</speed>
  <statistics>
   <discontinuity-time>..</discontinuity-time>
   <in-octets>41705747838</in-octets>
   <in-unicast-pkts>129128494</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>168135978</out-octets>
   <out-unicast-pkts>414310</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>
```

Outputs modified for screen display

YANG Model Deviations

```
module: ietf-ip
  augment /if:interfaces-state/if:interface:
    +--ro ipv4!
    |  +--ro forwarding?   boolean
    |  +--ro mtu?          uint16
    |  +--ro address* [ip]
    |  |  +--ro ip           inet:ipv4-address-no-zone
    |  |  +--ro (subnet)?
    |  |  |  +--:(prefix-length)
    |  |  |  |  +--ro prefix-length?   uint8
    |  |  |  +--:(netmask)
    |  |  |     +--ro netmask?     yang:dotted-quad
    |  |  +--ro origin?          ip-address-origin
```

Where is the <ipv4> leaf in the device data? ←

```
module cisco-xe-ietf-ip-deviation {
  namespace
    "http://cisco.com/ns/cisco-xe-ietf-ip-deviation";
  .
  .
  deviation /if:interfaces-state/if:interface/ip:ipv4{
      deviate not-supported;
      description  "Not supported in IOS-XE";
  }
}
```

YANG deviations allow for vendors to modify from standard models *when required*.

```
<interfaces-state xmlns="ietf-interfaces">
 <interface>
  <name>GigabitEthernet1</name>
  <type>ianaift:ethernetCsmacd</type>
  <admin-status>up</admin-status>
  <oper-status>up</oper-status>
  <last-change>2017-11-14T13:33:</last-change>
  <if-index>0</if-index>
  <phys-address>00:50:56:bb:c9:2c</phys-address>
  <speed>1024000</speed>
  <statistics>
   <discontinuity-time>..</discontinuity-time>
   <in-octets>41705747838</in-octets>
   <in-unicast-pkts>129128494</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>168135978</out-octets>
   <out-unicast-pkts>414310</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>
```

Outputs modified for screen display

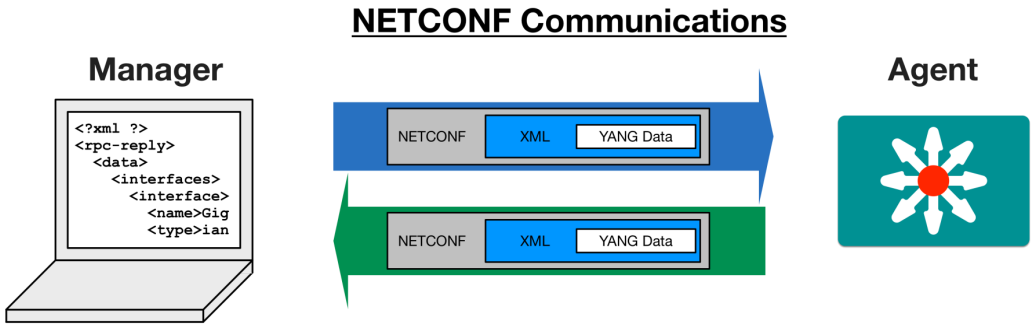
YANG Summary

YANG Summary

- YANG is a Data Modeling Language
- YANG Modules are constructed to create standard data models for network data
- YANG Data sent to or from a network device will be formatted in either XML or JSON depending on the protocol (ex: NETCONF or RESTCONF)

Understanding NETCONF

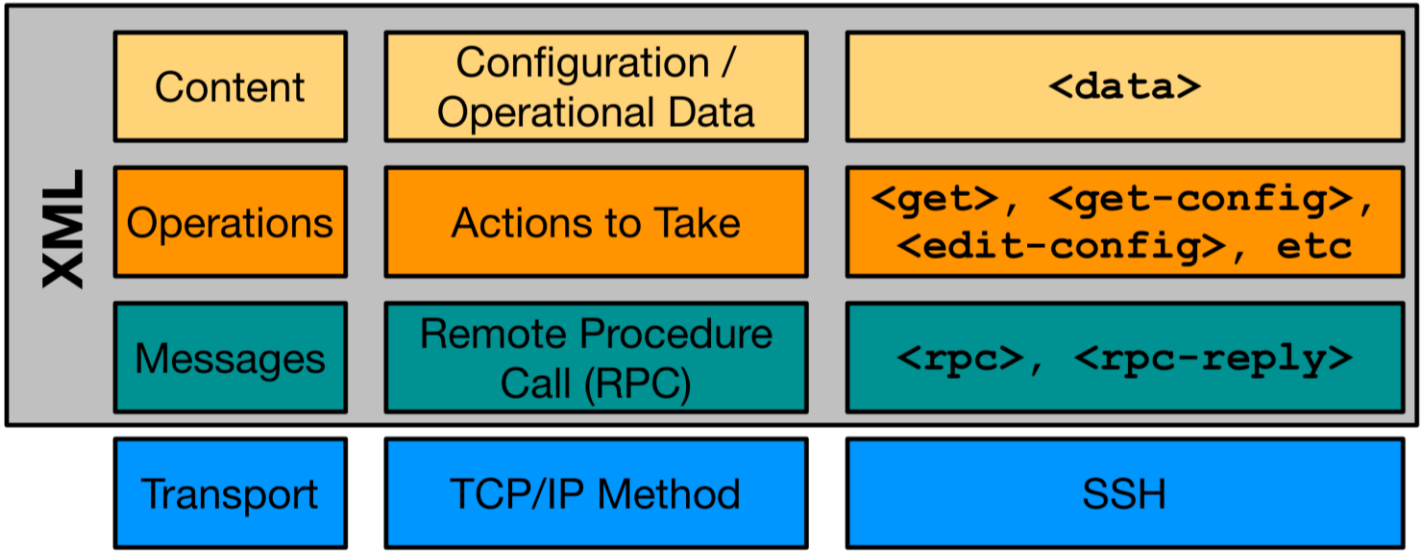
Introducing the NETCONF Protocol



Some key details:

- Initial standard in 2006 with RFC4741
- Latest standard is RFC6241 in 2011
- Does **NOT** explicitly define content

NETCONF Protocol Stack



Transport - SSH

```
$ ssh admin@192.168.0.1 -p 830 -s netconf
admin@192.168.0.1's password:
```

SSH Login

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
  <capability>urn:ietf:params:netconf:base:1.1</capability>
  <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
  <capability>urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring</capability>
  <capability>urn:ietf:params:xml:ns:yang:ietf-interfaces</capability>
  [output omitted and edited for clarity]
</capabilities>
<session-id>19150</session-id></hello>]]>]]>
```

Server (Agent) sends hello

```
<?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>
</capabilities>
  </hello>]]>]]>
```

Client (Manager) sends hello

Example edited for simplicity and brevity

Transport - SSH

```
$ ssh admin@192.168.0.1 -p 830 -s netconf
admin@192.168.0.1's password:

<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
  <capability>urn:ietf:params:netconf:base:1.1</capability>
  <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
  <capability>urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring</capability>
  <capability>urn:ietf:params:xml:ns:yang:ietf-interfaces</capability>
  [output omitted and edited for clarity]
</capabilities>
<session-id>19150</session-id></hello>]]>]]>

<?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>
</capabilities>
  </hello>]]>]]>
```

Don’t NETCONF Like this!

Example edited for simplicity and brevity

Messages - Remote Procedure Call (RPC)

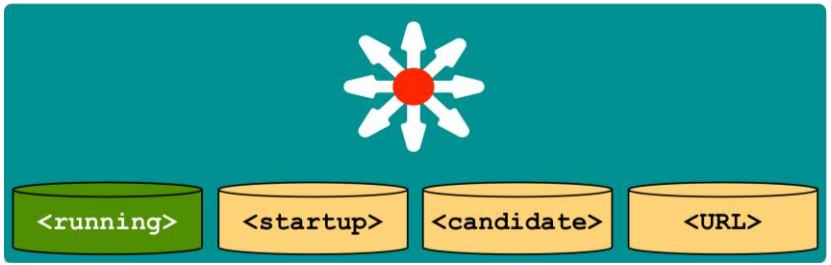
Operations - NETCONF Actions

Operation	Description
<get>	Retrieve running configuration and device state information
<get-config>	Retrieve all or part of specified configuration data store
<edit-config>	Loads all or part of a configuration to the specified configuration data store
<copy-config>	Replace an entire configuration data store with another
<delete-config>	Delete a configuration data store
<commit>	Copy candidate data store to running data store
<lock> / <unlock>	Lock or unlock the entire configuration data store system
<close-session>	Graceful termination of NETCONF session
<kill-session>	Forced termination of NETCONF session

Additional Operations

- The IETF isn't able to anticipate every use case from every vendor.
- Additional operations can be defined by the vendor
- Support extended through RPC call to vendor-specific YANG model

NETCONF Data Stores

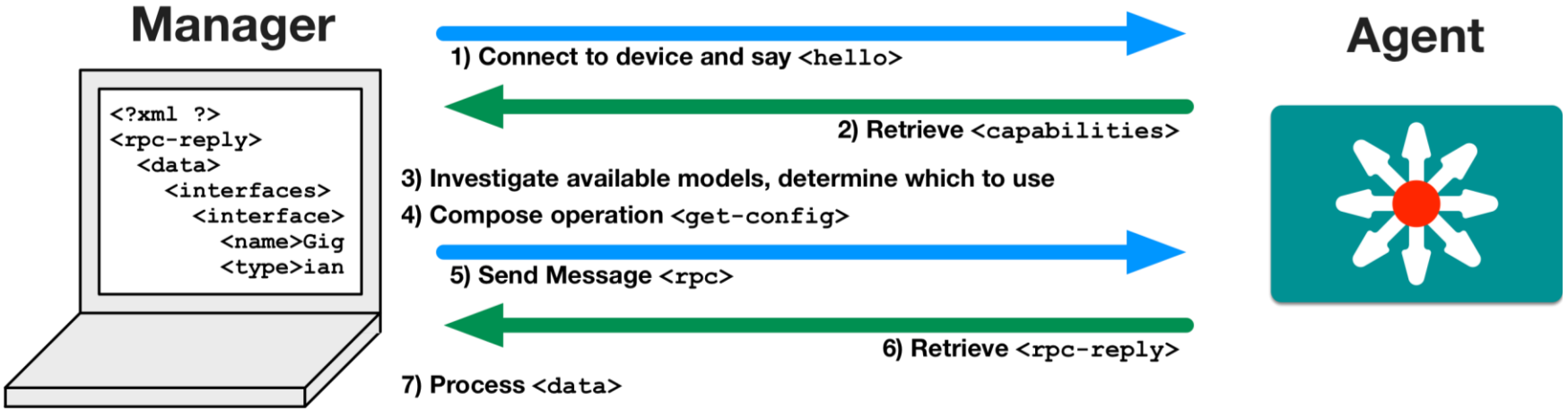


Data Store Key Points

- Entire or partial configuration
- "running" is the only mandatory data store
- Not all data stores are writeable
- A "URL" data store is supported by IOS to enable `<config-copy>`
- Every NETCONF message must target a data store

```
result = m.get_config('running')
```

NETCONF Communications



NETCONF in Code with Python

NETCONF and Python: ncclient

- Full NETCONF Manager implementation in Python
  - https://ncclient.readthedocs.io
- Simplifies connection and communication.
- Deals in raw XML

```
from ncclient import manager

m = manager.connect(host="192.168.0.1",
                    port=830,
                    username="admin",
                    password="cisco123",
                    hostkey_verify=False
                    )

m.close_session()
```

From: http://ncclient.readthedocs.io/en/latest/

Saying <hello> with Python and ncclient

- example1.py: Saying `<hello>`
- `manager.connect()` opens NETCONF session with device
  - Parameters: host & port, user & password
  - `hostkey_verify=False` Trust cert
- Stores capabilities

```
from device_info import ios_xe1
from ncclient import manager

if __name__ == '__main__':
    with manager.connect(host=ios_xe1["address"], port=ios_xe1["port"],
                         username=ios_xe1["username"],
                         password=ios_xe1["password"],
                         hostkey_verify=False) as m:

        print("Here are the NETCONF Capabilities")
        for capability in m.server_capabilities:
            print(capability)
```

BRKDEV-1368/netconf/device_info.py
BRKDEV-1368/netconf/example1.py

Understanding the Capabilities List

```
DevNet$ python example1.py
Here are the NETCONF Capabilities

urn:ietf:params:netconf:base:1.0
urn:ietf:params:netconf:base:1.1
.
urn:ietf:params:xml:ns:yang:ietf-interfaces?module=ietf-interfaces&revision=2014-05-08&features=pre-
provisioning,if-mib,arbitrary-names&deviations=ietf-ip-devs

http://cisco.com/ns/ietf-ip/devs?module=ietf-ip-devs&revision=2016-08-10

http://cisco.com/ns/yang/Cisco-IOS-XE-native?module=Cisco-IOS-XE-native&revision=2017-02-07
```

Example edited for simplicity and brevity

Two General Types

- Base NETCONF capabilities
- Data Models Supported

Understanding the Capabilities List

```
urn:ietf:params:xml:ns:yang:ietf-interfaces
  ? module=ietf-interfaces
  & revision=2014-05-08
  & features=pre-provisioning,if-mib,arbitrary-names
  & deviations=ietf-ip-devs
.
http://cisco.com/ns/ietf-ip/devs
  ? module=ietf-ip-devs
  & revision=2016-08-10
```

Example edited for simplicity and brevity

Data Model Details

- Model URI
- Module Name and Revision Date
- Protocol Features
- Deviations – Another model that modifies this one

Automate Your Network with NETCONF

Getting Interface Details with XML Filter

- example2.py: Retrieving info with ncclient
- Send <get> to retrieve config and state data
- Process and leverage XML within Python
- Report back current state of interface

```
from device_info import ios_xe1
from ncclient import manager
import xmltodict

# NETCONF filter to use
netconf_filter = open("filter-ietf-interfaces.xml").read()

if __name__ == '__main__':
    with manager.connect(host=ios_xe1["address"], port=ios_xe1["port"],
                         username=ios_xe1["username"],
                         password=ios_xe1["password"],
                         hostkey_verify=False) as m:

        # Get Configuration and State Info for Interface
        netconf_reply = m.get(netconf_filter)

        # Process the XML and store in useful dictionaries
        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"]))
        print("  Description: {}".format(intf_config["description"]))
        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"]))
```

BRKDEV-1368/netconf/example2.py
BRKDEV-1368/netconf/filter-ietf-interfaces.xml

Getting Interface Details with XML Filter

- example2.py: Retrieving info with ncclient
- Send <get> to retrieve config and state data
- Process and leverage XML within Python
- Report back current state of interface

```
<filter>
  <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>GigabitEthernet2</name>
    </interface>
  </interfaces>
  <interfaces-state xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>GigabitEthernet2</name>
    </interface>
  </interfaces-state>
</filter>
```

BRKDEV-1368/netconf/example2.py
BRKDEV-1368/netconf/filter-ietf-interfaces.xml

Getting Interface Details with XML Filter

- example2.py: Retrieving info with ncclient
- Send <get> to retrieve config and state data
- Process and leverage XML within Python
- Report back current state of interface

```
from device_info import ios_xe1
from ncclient import manager
import xmltodict

# NETCONF filter to use
netconf_filter = open("filter-ietf-interfaces.xml").read()

if __name__ == '__main__':
    with manager.connect(host=ios_xe1["address"], port=ios_xe1["port"],
                         username=ios_xe1["username"],
                         password=ios_xe1["password"],
                         hostkey_verify=False) as m:

        # Get Configuration and State Info for Interface
        netconf_reply = m.get(netconf_filter)

        # Process the XML and store in useful dictionaries
        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"]))
        print("  Description: {}".format(intf_config["description"]))
        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"]))
```

BRKDEV-1368/netconf/example2.py
BRKDEV-1368/netconf/filter-ietf-interfaces.xml

Getting Interface Details

```
DevNet$ python  example2.py

Interface Details:
  Name: GigabitEthernet1
  Description: DON'T TOUCH ME
  Type: ianaift:ethernetCsmacd
  MAC Address: 00:50:56:bb:74:d5
  Packets Input: 592268689
  Packets Output: 21839
```

BRKDEV-1368/netconf/example2.py

Getting Interface Details with XPath

- example_xpath.py: Retrieving info with ncclient and XPath
- Send <get> to retrieve and state data
- Process the data
- Report back current state of interface

```
from device_info import ios_xe1
from ncclient import manager
import xmltodict

if __name__ == '__main__':
    with manager.connect(host=ios_xe1["address"], port=ios_xe1["port"],
                         username=ios_xe1["username"],
                         password=ios_xe1["password"],
                         hostkey_verify=False) as m:

        # Get Configuration and State Info for Interface
        netconf_reply = m.get(filter=('xpath' ,
                                      "/interfaces-state/interface[name='GigabitEthernet1']"
                                      "/statistics/out-unicast-pkts"))

        intf_details = xmltodict.parse(netconf_reply.xml)["rpc-reply"]["data"]
        intf_info = intf_details["interfaces-state"]["interface"]

        print("")
        print("Interface Details:")
        print("  Name: {}".format(intf_info["name"]))
        print("  Packets Output: {}".format(intf_info["statistics"]["out-unicast-pkts"]))
```

BRKDEV-1368/netconf/example_xpath.py

Getting Interface Details

```
DevNet$python example_xpath.py

Interface Details:
  Name: GigabitEthernet1
  Packets Output: 415200
```

BRKDEV-1368/netconf/example_xpath.py

Configuring Interface Details

- example3.py: Editing configuration with ncclient
- Constructing XML Config Payload for NETCONF
- Sending <edit-config> operation with ncclient
- Verify result

```
from device_info import ios_xe1
from ncclient import manager

# NETCONF Config Template to use
netconf_template = open("config-temp-ietf-interfaces.xml").read()

if __name__ == '__main__':
    # Build the XML Configuration to Send
    netconf_payload = netconf_template.format(int_name="GigabitEthernet2",
                                              int_desc="Configured by NETCONF",
                                              ip_address="10.255.255.1",
                                              subnet_mask="255.255.255.0"
                                              )
    print("Configuration Payload:")
    print("----------------------")
    print(netconf_payload)

    with manager.connect(host=ios_xe1["address"], port=ios_xe1["port"],
                         username=ios_xe1["username"],
                         password=ios_xe1["password"],
                         hostkey_verify=False) as m:

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

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

BRKDEV-1368/netconf/example3.py

Configuring Interface Details

- example3.py: Editing configuration with ncclient
- Constructing XML Config Payload for NETCONF
- Sending <edit-config> operation with ncclient
- Verify result

config-temp-ietf-interfaces.xml

```
<config>
  <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>{int_name}</name>
      <description>{int_desc}</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">
        <address>
          <ip>{ip_address}</ip>
          <netmask>{subnet_mask}</netmask>
        </address>
      </ipv4>
    </interface>
  </interfaces>
</config>
```

```
netconf_template = open("config-temp-ietf-interfaces.xml").read()

if __name__ == '__main__':
    # Build the XML Configuration to Send
    netconf_payload = netconf_template.format(int_name="GigabitEthernet2",
                                              int_desc="Configured by NETCONF",
                                              ip_address="10.255.255.1",
                                              subnet_mask="255.255.255.0"
                                              )
    print("Configuration Payload:")
    print("----------------------")
    print(netconf_payload)
```

BRKDEV-1368/netconf/config-temp-ietf-interfaces.xml

BRKDEV-1368/netconf/example3.py

Configuring Interface Details

- example3.py: Editing configuration with ncclient
- Constructing XML Config Payload for NETCONF
- Sending <edit-config> operation with ncclient
- Verify result

```
from device_info import ios_xe1
from ncclient import manager

# NETCONF Config Template to use
netconf_template = open("config-temp-ietf-interfaces.xml").read()

if __name__ == '__main__':
    # Build the XML Configuration to Send
    netconf_payload = netconf_template.format(int_name="GigabitEthernet2",
                                              int_desc="Configured by NETCONF",
                                              ip_address="10.255.255.1",
                                              subnet_mask="255.255.255.0"
                                              )
    print("Configuration Payload:")
    print("----------------------")
    print(netconf_payload)

    with manager.connect(host=ios_xe1["address"], port=ios_xe1["port"],
                         username=ios_xe1["username"],
                         password=ios_xe1["password"],
                         hostkey_verify=False) as m:

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

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

BRKDEV-1368/netconf/config-temp-ietf-interfaces.xml
BRKDEV-1368/netconf/example3.py

Configuring Interface Details

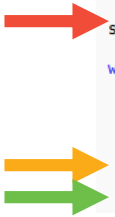
```
DevNet$ python -i example3.py
Configuration Payload:
----------------------
<config>
  <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
          <interface>
            <name>GigabitEthernet2</name>
            <description>Configured by NETCONF</description>
            <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">
                  ianaift:ethernetCsmacd
          </type>
            <enabled>true</enabled>
            <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
                    <address>
                      <ip>10.255.255.1</ip>
                      <netmask>255.255.255.0</netmask>
                    </address>
            </ipv4>
          </interface>
  </interfaces>
</config>

<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns=”urn.." message-id=”..9784" xmlns:nc="urn..">
  <ok/>
</rpc-reply>
```

Example edited for simplicity and brevity

BRKDEV-1368/netconf/config-temp-ietf-interfaces.xml
BRKDEV-1368/netconf/example3.py

Saving Running Configuration with ncclient

- example_save_rpc.py: Save running configuration with ncclient
- Constructing XML Config Payload for NETCONF
- Sending custom save operation with ncclient
- Verify result



```
from ncclient import manager, xml_
from device_info import *

save_body = '<cisco-ia:save-config xmlns:cisco-ia="http://cisco.com/yang/cisco-ia"/>'

with manager.connect(host=ios_xe1["address"], port=ios_xe1["port"],
                     username=ios_xe1["username"],
                     password=ios_xe1["password"],
                     hostkey_verify=False) as m:

    save_rpc = m.dispatch(xml_.to_ele(save_body))
    print(save_rpc)
```

BRKDEV-1368/netconf/example_save_rpc.py

Saving Running Configuration with ncclient

```
<?xml version="1.0" encoding="UTF-8"?>
   <rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
              message-id="urn:uuid:d12c3cc5-f638-499e-9e57-c8d2402fdfeb"
              xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
     <result xmlns='http://cisco.com/yang/cisco-ia'>
              Save running-config successful
     </result>
  </rpc-reply>
```

NETCONF Summary

NETCONF Summary

- The elements of the NETCONF transport protocol
- How to leverage ncclient to use NETCONF in Python
- Examples retrieving and configuring data from a NETCONF Agent

Understanding RESTCONF

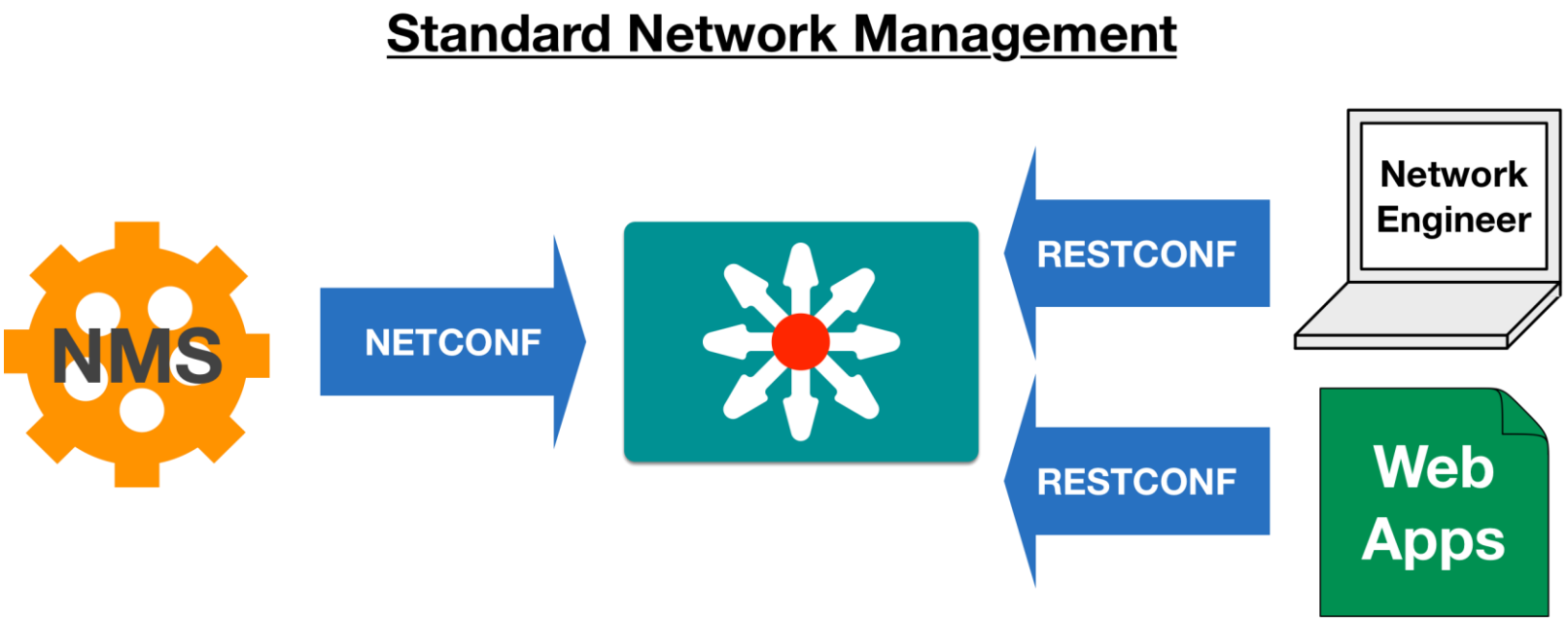
RESTCONF Details

"an HTTP-based protocol that provides a programmatic interface for accessing data defined in YANG…"

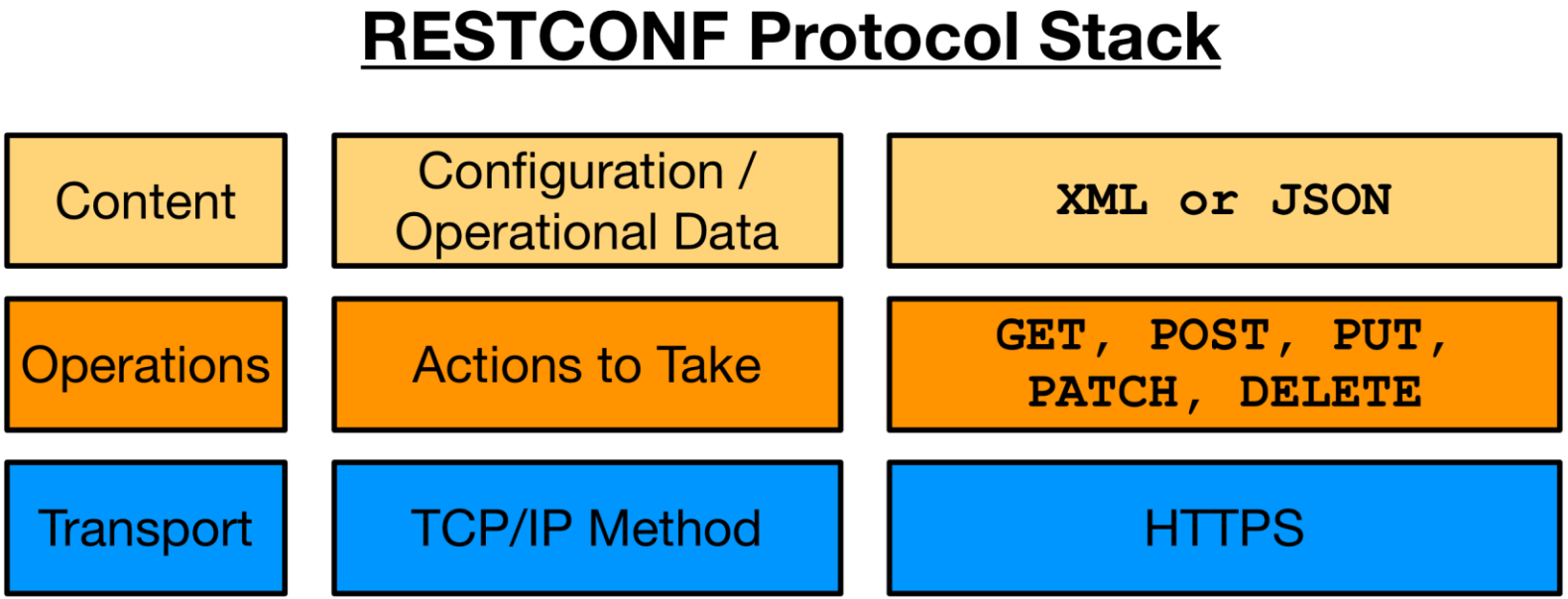
- RFC 8040 - January 2017
- Uses HTTPS for transport
- Tightly coupled to the YANG data model definitions
- Provides JSON or XML data formats

- https://tools.ietf.org/html/rfc8040

What about NETCONF?



RESTCONF Protocol Stack & Transport



Operations - HTTP CRUD

RESTCONF	NETCONF
GET	<get> , <get-config>
POST	<edit-config> (operation="create")
PUT	<edit-config> (operation="create/replace")
PATCH	<edit-config> (operation="merge")
DELETE	<edit-config> (operation="delete")

Content - XML or JSON

HTTP Headers

- **Content-Type**: Specify the type of data being sent from the client
- **Accept**: Specify the type of data being requested by the client

RESTCONF MIME Types

- **application/yang-data+json**
- **application/yang-data+xml**

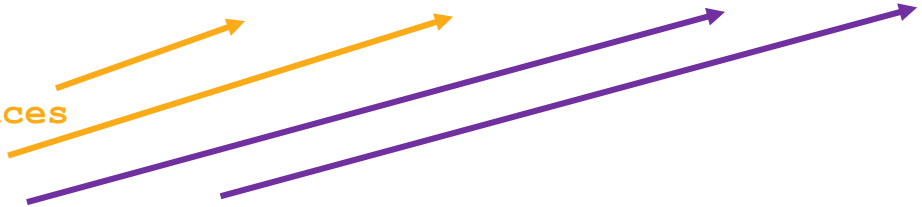
Constructing RESTCONF URIs for Data Resources

`https://<ADDRESS>/<ROOT>/data/<[YANG MODULE:]CONTAINER>/<LEAF>[?<OPTIONS>]`

- **ADDRESS** - Of the RESTCONF Agent
- **ROOT** - The main entry point for RESTCONF requests.
  Discoverable at `https://<ADDRESS>/.well-known/host-meta`
- **data** - The RESTCONF API resource type for data
  - *The "operations" resource type used to access RPC operations available*
- **[YANG MODULE:]CONTAINER** - The base model container being used. Providing the module name is optional.
- **LEAF** - An individual element from within the container
- **[?<OPTIONS>]** - optional parameters that impact returned results.

URL Creation Review

```
https://<ADDRESS>/restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet1?depth=unbounded
```



```
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
```

Key:
https://<ADDRESS>/<ROOT>/data>/<[YANG MODULE:]CONTAINER>/<LEAF>[?<OPTIONS>]

Options Examples:

- `depth=unbounded`
  Follow nested models to end. Integer also supported
- `content=[`**`all`**`, config, nonconfig]`
  Query option controls type of data returned.
- `fields=`***`expr`***
  Limit what leafs are returned

Using RESTCONF with Postman

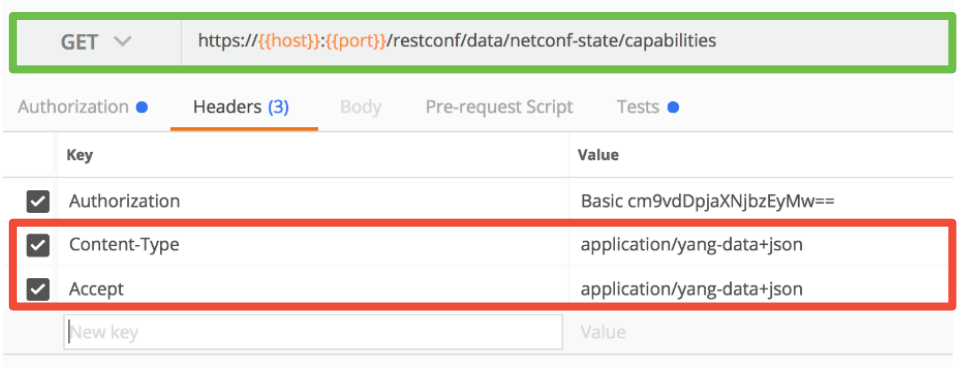
Postman: Powerful but Simple REST API Client

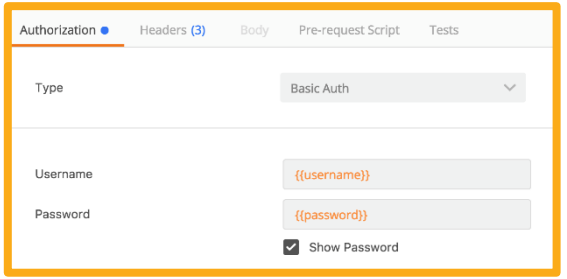
- Quickly test APIs in GUI
- Save APIs into Collections for reuse
- Manage multiple environments
- Auto generate code from API calls
- Standalone Application or Chrome Plugin

https://www.getpostman.com

Step 1: Get Capabilities List via RESTCONF

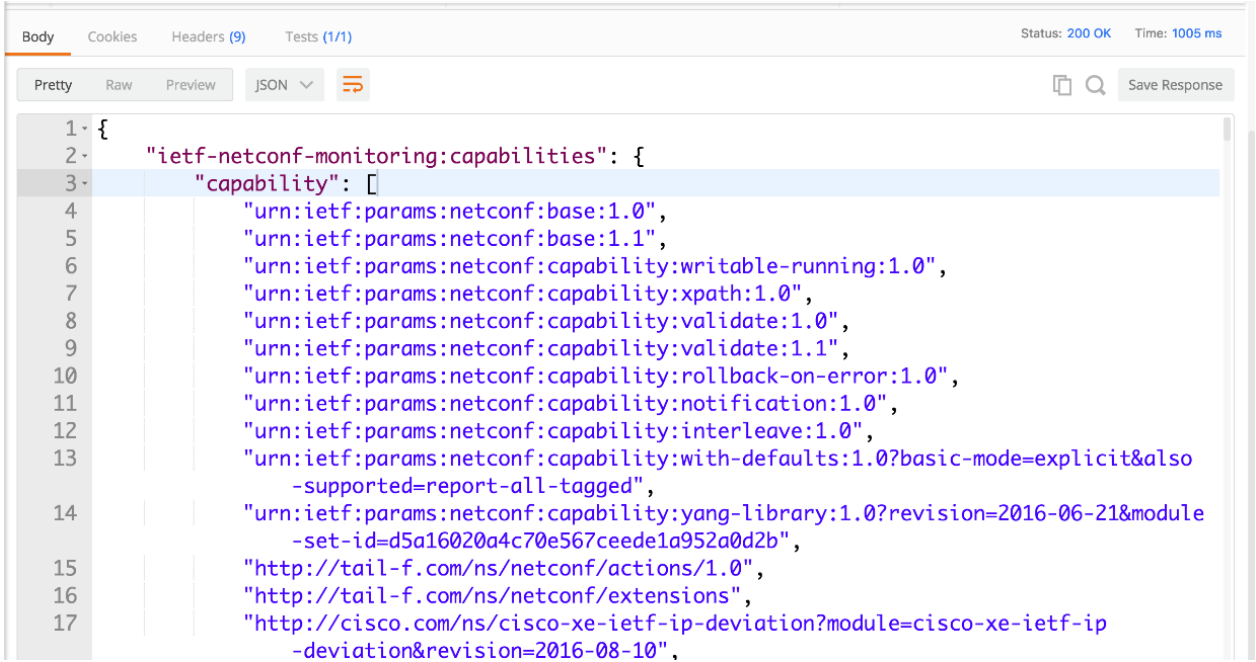
- **GET**
  `/restconf/data/netconf-state/capabilities`
- Add RESTCONF Headers
  - **Content-Type** and **Accept**
    `application/yang-data+json`
    (or xml)
- Configure Basic Auth with username and password variables





Step 1: Get Capabilities List via RESTCONF

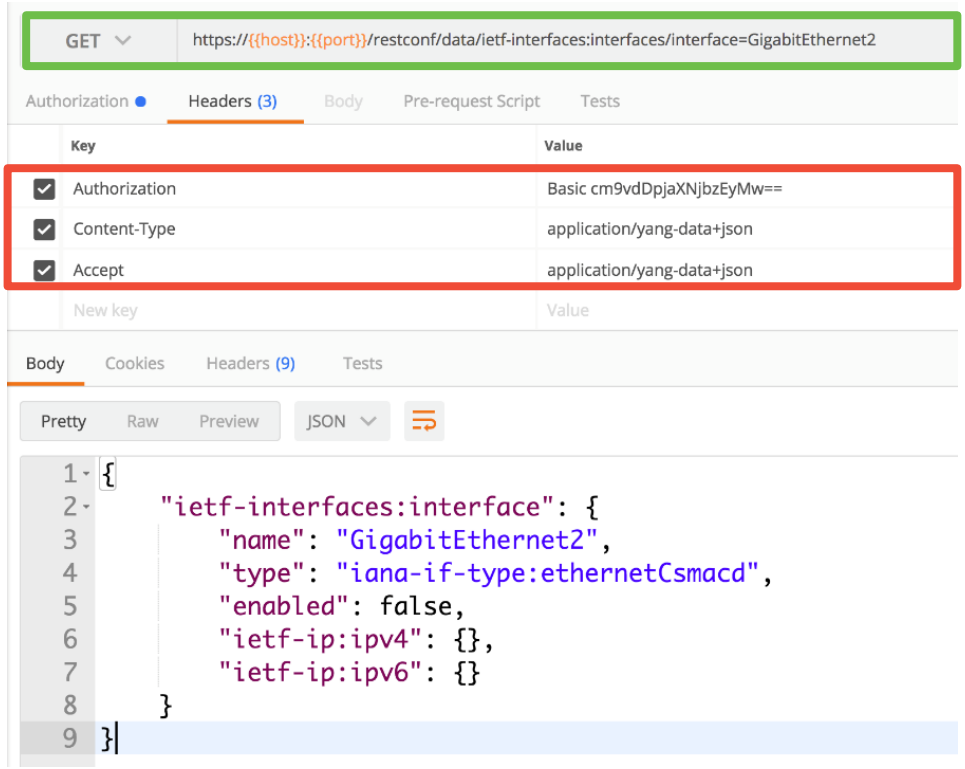
- Send and review results



Automate Your Network with RESTCONF

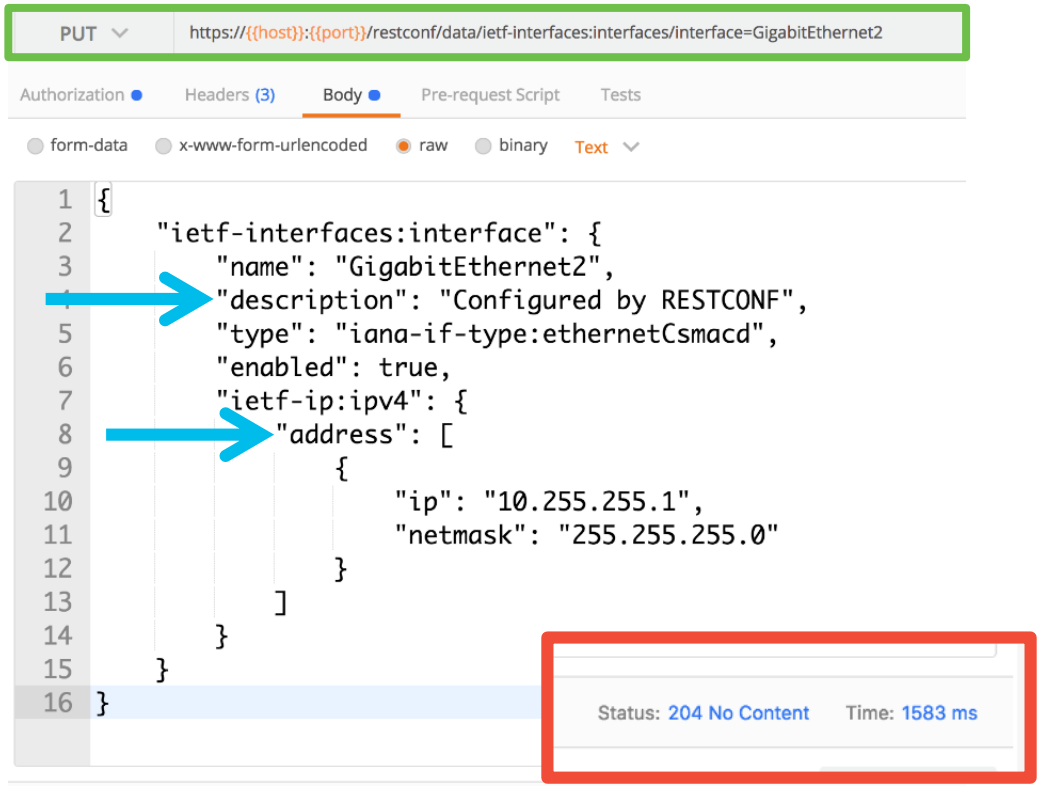
Getting Interface Details

- **GET**
  `restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet2`
- Configure Auth and Headers



Configuring Interface Details

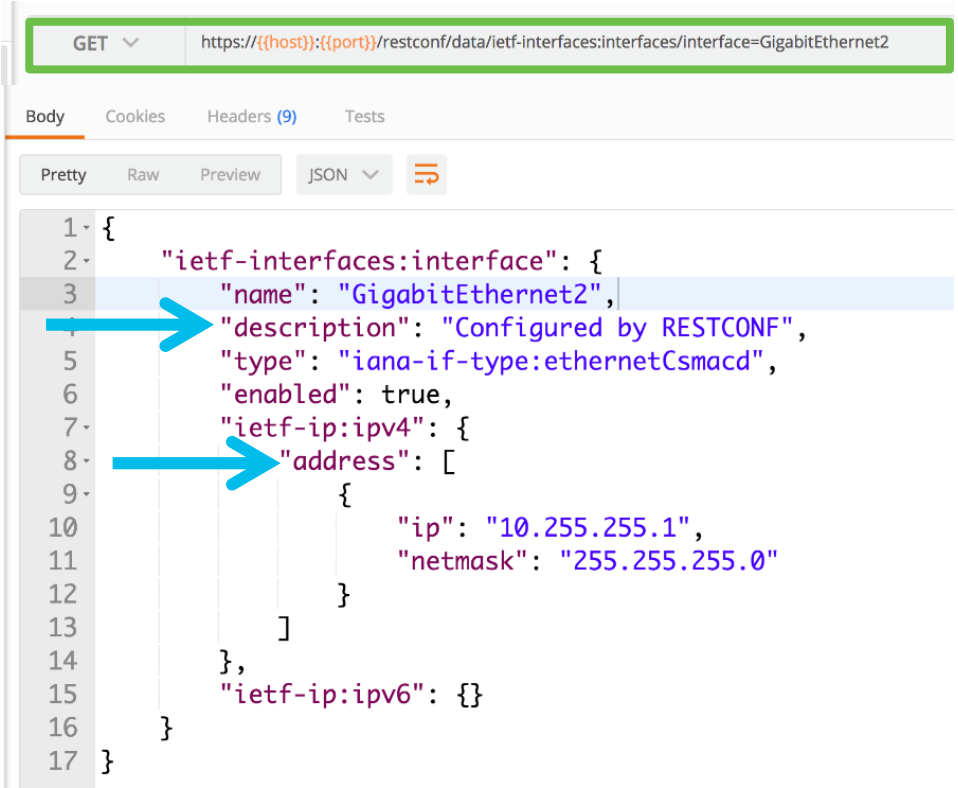
- **PUT**
  `restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet2`
- Configure Auth and Headers
- Configure Body (raw)
- Send and check status code



Configuring Interface Details - Verification

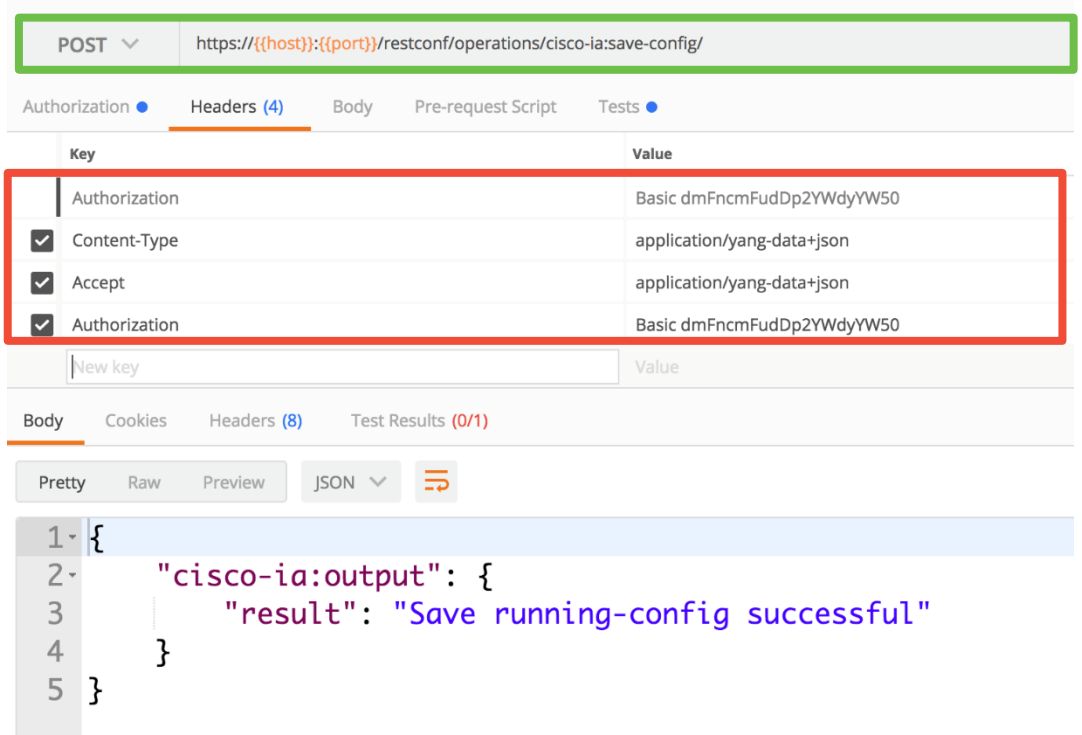
- **GET**
  `restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet2`

- Configure Auth and Headers
- Check that the new config was successful



Save Running Configuration to Startup RPC Operation

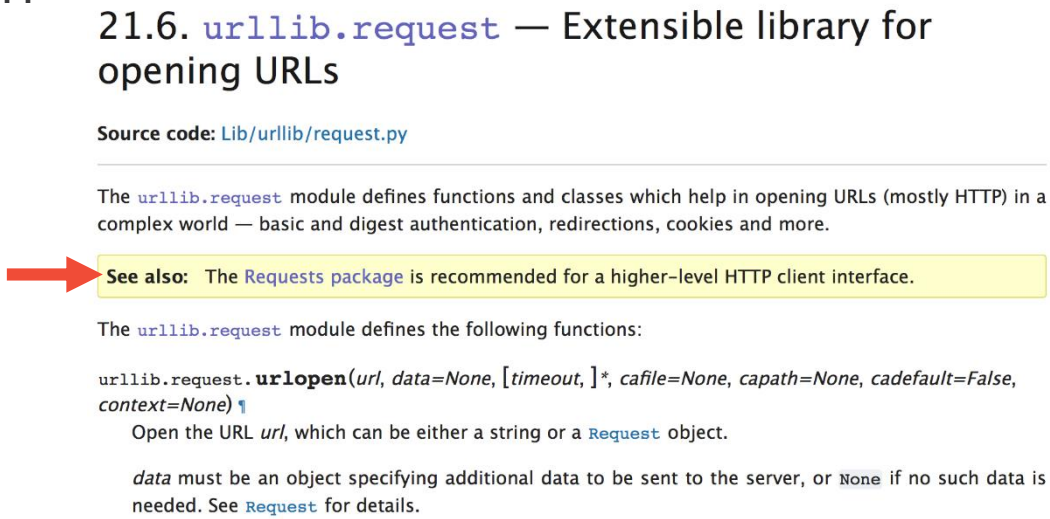
- **POST**
  `restconf/operations/cisco-ia:save-config/`
- Configure Auth and Headers



RESTCONF with Python

Python Libraries for RESTCONF

- Treat like other “REST” and HTTP APIs
- Core Python Library
  - urllib (urllib.request)
- Other HTTP Library
  - Requests: http for humans



21.6. `urllib.request` — Extensible library for opening URLs

Source code: Lib/urllib/request.py

The `urllib.request` module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

See also: The Requests package is recommended for a higher-level HTTP client interface.

The `urllib.request` module defines the following functions:

`urllib.request.`**`urlopen`**(*url*, *data=None*, [*timeout*,]*, *cafile=None*, *capath=None*, *cadefault=False*, *context=None*) ¶

Open the URL *url*, which can be either a string or a `Request` object.

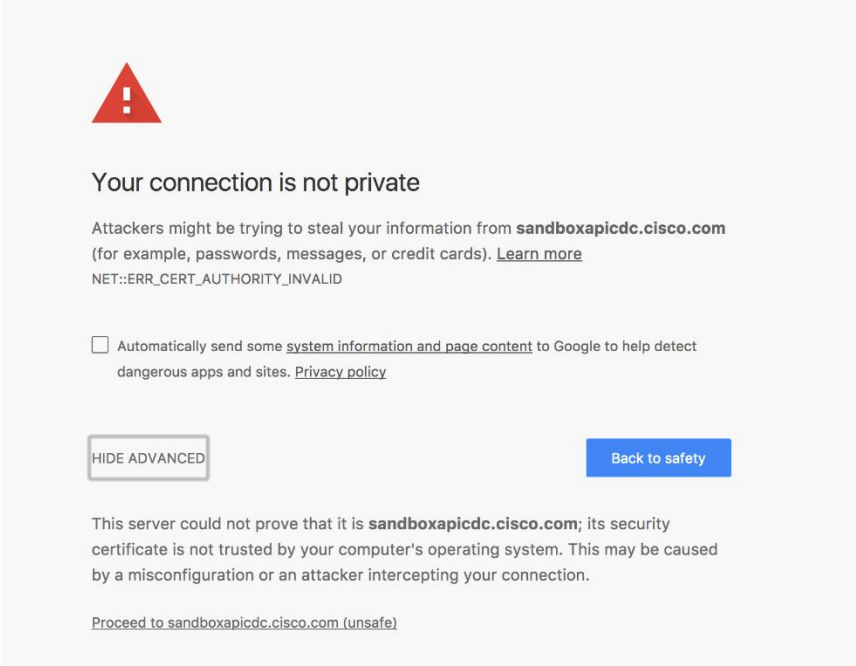
data must be an object specifying additional data to be sent to the server, or `None` if no such data is needed. See `Request` for details.

RESTCONF and HTTPS/SSL

- Per RFC8040 RESTCONF requires HTTPS for security
- HTTPS leverages SSL/TLS certificates to
  - Encrypt data transmitted between server and client
  - Verify a sites identity
- Encryption accomplished through Private/Public keys in Certificate
- Identity is all about “trust” and the “signer” of a certificate

Self-Signed SSL Certificates are Common

- No problems with encryption
- “Trust” fails due to no Certificate Authority
- Common in most of our daily lives
  - Network web GUIs
  - Internal corporate web sites
  - Poorly managed company sites
- Knee-jerk reaction to click “Accept” or “Proceed”



“Accepting” Self-Signed Certificates with Python Requests

- Requests (via urllib) performs SSL Validation by **default**

```
>>> import requests
>>> url = "https://ios-xe-mgmt.cisco.com:9443/.well-known/host-meta"

>>> response = requests.get(url, auth=("root", "D_Vay!_10&"))

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    response = requests.get(url, auth=("root", "D_Vay!_10&"))
SSLError: HTTPSConnectionPool(host='ios-xe-mgmt.cisco.com', port=9443): Max
retries exceeded with url: /.well-known/host-meta (Caused by
SSLError(SSLError(1, u'[SSL: CERTIFICATE_VERIFY_FAILED] certificate verify
failed (_ssl.c:661)'),))
```

https://github.com/CiscoDevNet/BRKDEV-1368/blob/master/demo/layer3_interface_config.j2

“Accepting” Self-Signed Certificates with Python Requests

- Disable with `verify = False`
- Still generates a Warning notification
- But the request is successful

```
>>> import requests
>>> url = "https://ios-xe-mgmt.cisco.com:9443/.well-known/host-meta"

>>> response = requests.get(url, auth=("root", "D_Vay!_10&"), verify=False)

Warning (from warnings module):
  File "/Users/hapresto/coding/BRKDEV-1368/venv/lib/python2.7/site-
packages/urllib3/connectionpool.py", line 858
    InsecureRequestWarning)
InsecureRequestWarning: Unverified HTTPS request is being made. Adding
certificate verification is strongly advised. See:
https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
```

```
>>> print(response.text)

<XRD xmlns='http://docs.oasis-open.org/ns/xri/xrd-1.0'>
    <Link rel='restconf' href='/restconf'/>
</XRD>
```

Disabling Insecure SSL Warnings from urllib

- Import urllib
- Disable specific warning

```
>>> import urllib3
>>> urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
>>> response = requests.get(url, auth=("root", "D_Vay!_10&"), verify=False)
```

```
>>> print(response.text)

<XRD xmlns='http://docs.oasis-open.org/ns/xri/xrd-1.0'>
    <Link rel='restconf' href='/restconf'/>
</XRD>
```

RESTCONF Summary

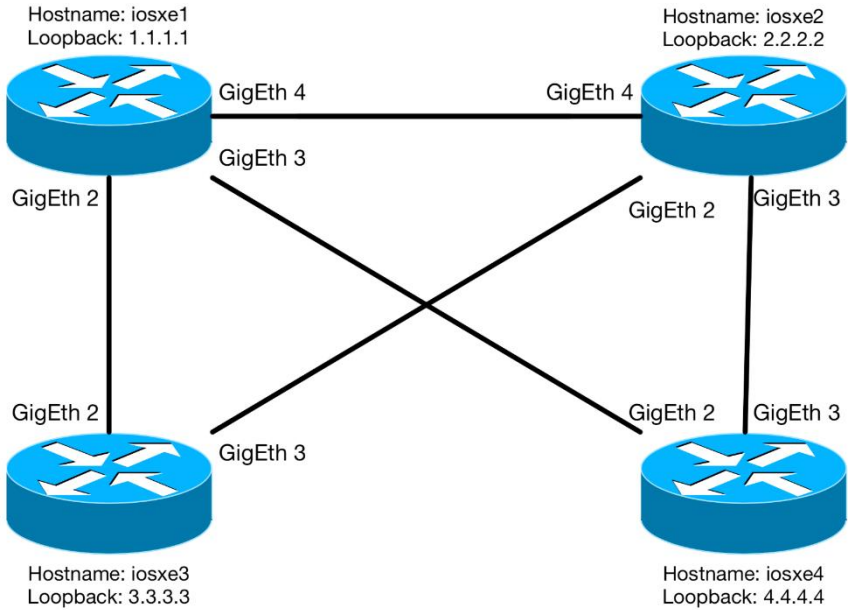
Review

- The elements of the RESTCONF transport protocol
- How to leverage Postman to use RESTCONF
- Examples retrieving and configuring data using RESTCONF
- Considerations of using RESTCONF with Python

Model Driven Programmability in Action!

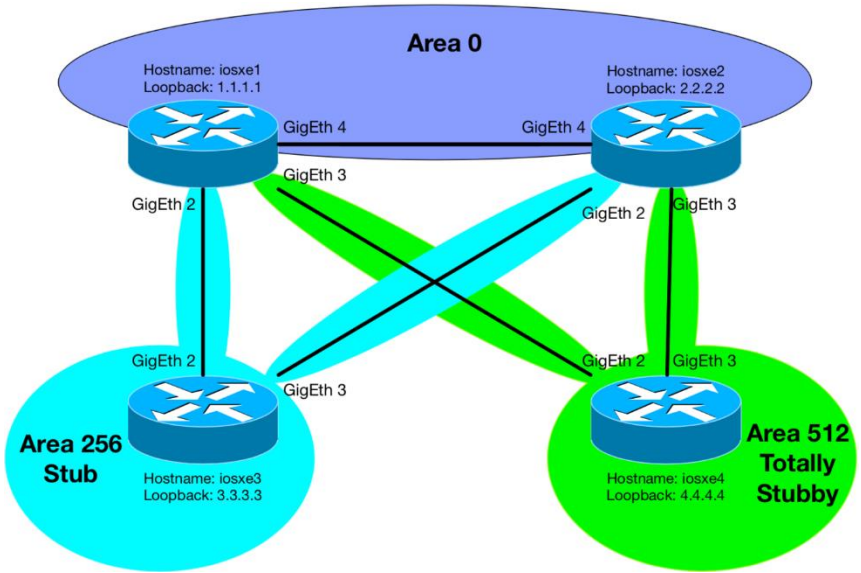
Starting Network Topology

- Physical Topology
  - IOS XE Routers
- Network has been cabled already
- Management access to devices enabled
  - No other configuration completed



Desired Network Configuration

- Layer 3 Links between Core/Dist
  - Routed /30 networks
- OSPF Configuration
  - Multi-Area
  - 1 Stub and 1 Totally Stubby Area
  - Several connected networks advertised at each router



Use our NETCONF / YANG and Python Mastery!

- Standard YANG Models
- Native YANG Models
- Reusable XML Templates
- Store network configuration details in configuration file
- Use Python + ncclient to send configurations

YANG Model: openconfig-interfaces.yang

- Standard model for “managing network interfaces and subinterfaces”
- Augmented by:
  - openconfig-if-ip.yang
  - openconfig-if-ethernet.yang

```
module: openconfig-interfaces
    +--rw interfaces
       +--rw interface* [name]
          +--rw name                -> ../config/name
          +--rw config
          |  +--rw name?            string
          |  +--rw type             identityref
          |  +--rw mtu?             uint16
          |  +--rw description?     string
          |  +--rw enabled?         boolean
          +--rw subinterfaces
             +--rw subinterface* [index]
                +--rw index       -> ../config/index
                +--rw config
                |  +--rw index?           uint32
                |  +--rw description?     string
                |  +--rw enabled?         boolean
```

* Output edited for display on slide

https://github.com/openconfig/public/blob/master/release/models/interfaces

YANG Model: Cisco-IOS-XE-ospf.yang

- Native model for managing OSPF configuration and state
- Augments the Cisco-IOS-Native.yang model

```
module: Cisco-IOS-XE-ospf
  augment /ios:native/ios:router:
    +--rw ospf* [id]
       +--rw id                          uint16
       +--rw vrf?                        string
       +--rw area* [id]
       |  +--rw id          ios-types:ospf-area-type
       |  +--rw authentication!
       |  |  +--rw message-digest?   empty
       |  +--rw stub!
       |  |  +--rw no-ext-capability?   empty
       |  |  +--rw no-summary?          empty
       +--rw network* [ip mask]
       |  +--rw ip      inet:ipv4-address
       |  +--rw mask    inet:ipv4-address
       |  +--rw area?   ios-types:ospf-area-type
```

* Output edited for display on slide

https://github.com/YangModels/yang/tree/master/vendor/cisco/xe/1662

Creating Reusable Templates

- Jinja2 Templating Language http://jinja.pocoo.org
  - Powerful templating language
  - Variable insertion, conditionals, loops
  - Not just for Python
- Architects/Designers/Engineers create standard configurations
  - Combined with specific environment details = full configurations

Layer 3 Configuration Template

```
<config>
  <interfaces xmlns="http://openconfig.net/yang/interfaces">
    {% for interface in interfaces %}
    <interface>
      <name>{{interface.name}}</name>
      <config>
        <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">{{interface.type}}</type>
        <name>{{interface.name}}</name>
        <enabled>{{interface.enabled}}</enabled>
      </config>
      <subinterfaces>
        <subinterface>
          <index>0</index>
          <config>
            <index>0</index>
            <name>{{interface.name}}</name>
            <enabled>{{interface.enabled}}</enabled>
          </config>
          <ipv4 xmlns="http://openconfig.net/yang/interfaces/ip">
            <addresses>
              <address>
                <ip>{{interface.ip}}</ip>
                <config>
                  <ip>{{interface.ip}}</ip>
                  <prefix-length>{{interface.prefix}}</prefix-length>
                </config>
              </address>
            </addresses>
          </ipv4>
        </subinterface>
      </subinterfaces>
    </interface>
    {% endfor %}
  </interfaces>
</config>
```

* Output edited for display on slide

https://github.com/CiscoDevNet/BRKDEV-1368/blob/master/demo/layer3_interface_config.j2

Creating Reusable Templates

Layer 3 Configuration Template

Partial for Discussion

```
<config>
  <interfaces xmlns="http://openconfig.net/yang/interfaces">
    {% for interface in interfaces %}
    <interface>
      <name>{{interface.name}}</name>
      <config>
        <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-type">
          {{interface.type}}
        </type>
        <name>{{interface.name}}</name>
        <enabled>{{interface.enabled}}</enabled>
      </config>
    </interface>
    {% endfor %}
  </interfaces>
</config>
```

* Output edited for display on slide

https://github.com/CiscoDevNet/BRKDEV-1368/blob/master/demo/layer3_interface_config.j2

Creating Reusable Templates

OSPF Configuration Template

```
<ospf xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ospf">
  <id>{{ospf.process_id}}</id>
  {% for area in ospf.areas %}
  <area>
    <id>{{area.area}}</id>
    {% if area.type == "stub" %}
    <stub>
      {% if area.no_summary %}
      <no-summary />
      {% endif %}
    </stub>
    {% endif %}
  </area>
  {% endfor %}
  <router-id>{{ospf.router_id}}</router-id>
  {% for network in ospf.networks %}
  <network>
    <ip>{{network.network}}</ip>
    <mask>{{network.wildcard}}</mask>
    <area>{{network.area}}</area>
  </network>
  {% endfor %}
</ospf>
```

* Output edited for display on slide

https://github.com/CiscoDevNet/BRKDEV-1368/blob/master/demo/ospf_config.j2

Environment Specific Network Configuration Details

- 12 Factor Principal - *"strict separation of config from code"* https://www.12factor.net/config
- "Config" is everything that varies between device deployments
  - Examples: IPs, names, interfaces
- Your code should "read in" config from another source
  - CMDB, IPAM, data base, configuration file, etc
- YAML data format used in demo
  - Human readable data format used by many orchestration tools

```yaml
devices:
  # IOS XE 1
  - name: iosxe1
    interfaces:
      - name: GigabitEthernet3
        enabled: "true"
        ip: 10.0.0.5
        prefix: 30
      - name: Loopback101
        enabled: "true"
        ip: 192.168.8.1
        prefix: 24
    ospf:
      router_id: 1.1.1.1
      areas:
        - area: "256"
          type: stub
      networks:
        - network: 10.0.0.1
          wildcard: 0.0.0.0
          area: "256"
```

* Output edited for display on slide

https://github.com/CiscoDevNet/BRKDEV-1368/blob/master/demo/config_details.yaml

Sending Network Configurations with ncclient

Part 1: Loading configuration and templates

- PyYAML library used to process configuration
- Jinja2 Template objects created for each XML template

```
# Load Network Config Details from YAML Config File
print("Loading Network Configuration Details from YAML")
with open("config_details.yaml") as f:
    config = yaml.load(f.read())

# Create Jinja Template Objects for NETCONF Payloads
print("Setting Up NETCONF Templates")
# Layer 3 Interface Configurations
with open("layer3_interface_config.j2") as f:
    l3_template = Template(f.read())

# OSPF Routing Configuration
with open("ospf_config.j2") as f:
    ospf_template = Template(f.read())
```

https://github.com/CiscoDevNet/BRKDEV-1368/blob/master/demo/push_configs.py

Sending Network Configurations with ncclient

Part 2: Creating Device Specific Configurations

- "for" loop used across devices
- Combine device configuration details with templates
  - Save configs to text files

```
# Loop over network devices to create and deploy network config
print("Processing Device Configurations")
for device in config["devices"]:
    print("Device: {}".format(device["name"]))
    # Create Device Specific Configurations
    print("  Creating Device Specific Configurations from Templates")
    with open("netconf_configs/{}_layer3.cfg".format(device["name"]), "w") as f:
        l3_config = l3_template.render(interfaces=device["interfaces"])
        f.write(l3_config)
    with open("netconf_configs/{}_ospf.cfg".format(device["name"]), "w") as f:
        ospf_config = ospf_template.render(ospf=device["ospf"])
        f.write(ospf_config)
```

https://github.com/CiscoDevNet/BRKDEV-1368/blob/master/demo/push_configs.py

Sending Network Configurations with ncclient

Part 3: Connect to Device and Send Configurations

- ncclient.manager used to open single connection
- Use <edit-config> to send device specific payload
- Print RPC result for status

```
# Connect to Device with NETCONF
print("  Connecting to device with NETCONF")
with manager.connect(host=config["network_mgmt_host"],
                     port=device["netconf_port"],
                     username=config["username"],
                     password=config["password"],
                     hostkey_verify=False,
                     device_params={'name': 'default'},
                     allow_agent=False,
                     look_for_keys=False) as m:

    # Send NETCONF Configurations with <edit-config> RPC
    print("  Sending NETCONF Configuration edit-config operations")
    l3_resp = m.edit_config(l3_config, target = "running")
    ospf_resp = m.edit_config(ospf_config, target = "running")

    # Process XML data in replies
    l3_reply = xmltodict.parse(l3_resp.xml)
    ospf_reply = xmltodict.parse(ospf_resp.xml)

    # Print Config Replies
    print("    Layer 3 Interface Config: {}".format(l3_reply["rpc-reply"].keys()[3]))
    print("    OSPF Config: {}".format(ospf_reply["rpc-reply"].keys()[3]))
    print("")
```

https://github.com/CiscoDevNet/BRKDEV-1368/blob/master/demo/push_configs.py

Let’s see it in action!

Model Driven Programmability in Real Life

That was cool… but is that really the way networks will be configured in the future?

NETCONF/YANG Used in Other Tools

Management & Orchestration	Configuration Management	Network Controllers	Software Libraries
Cisco NSO	ANSIBLE	OPEN DAYLIGHT	YANG Development Kit (YDK)
	puppet labs	CISCO DNA CENTER	
		APIC-EM	NAPALM

Questions?

Review

- The Road to Model Driven Programmability
- Introduction to YANG Data Models
- Introduction to NETCONF
- Introduction to RESTCONF
- Model Driven Programmability in Action
- Model Driven Programmability in Real Life
- Conclusion and Q/A

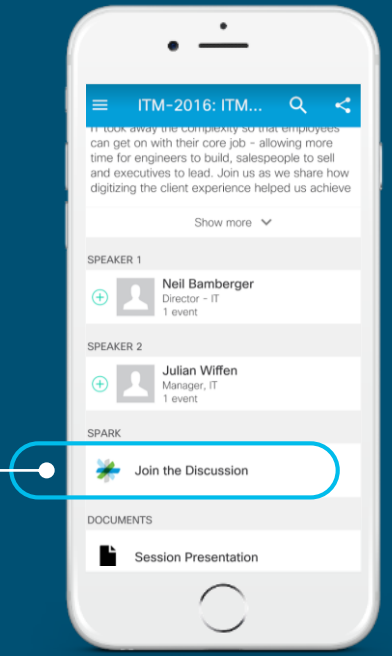
Cisco Spark

Questions?

Use Cisco Spark to communicate with the speaker after the session

How

1. Find this session in the Cisco Live Mobile App
2. Click “Join the Discussion”
3. Install Spark or go directly to the space
4. Enter messages/questions in the space

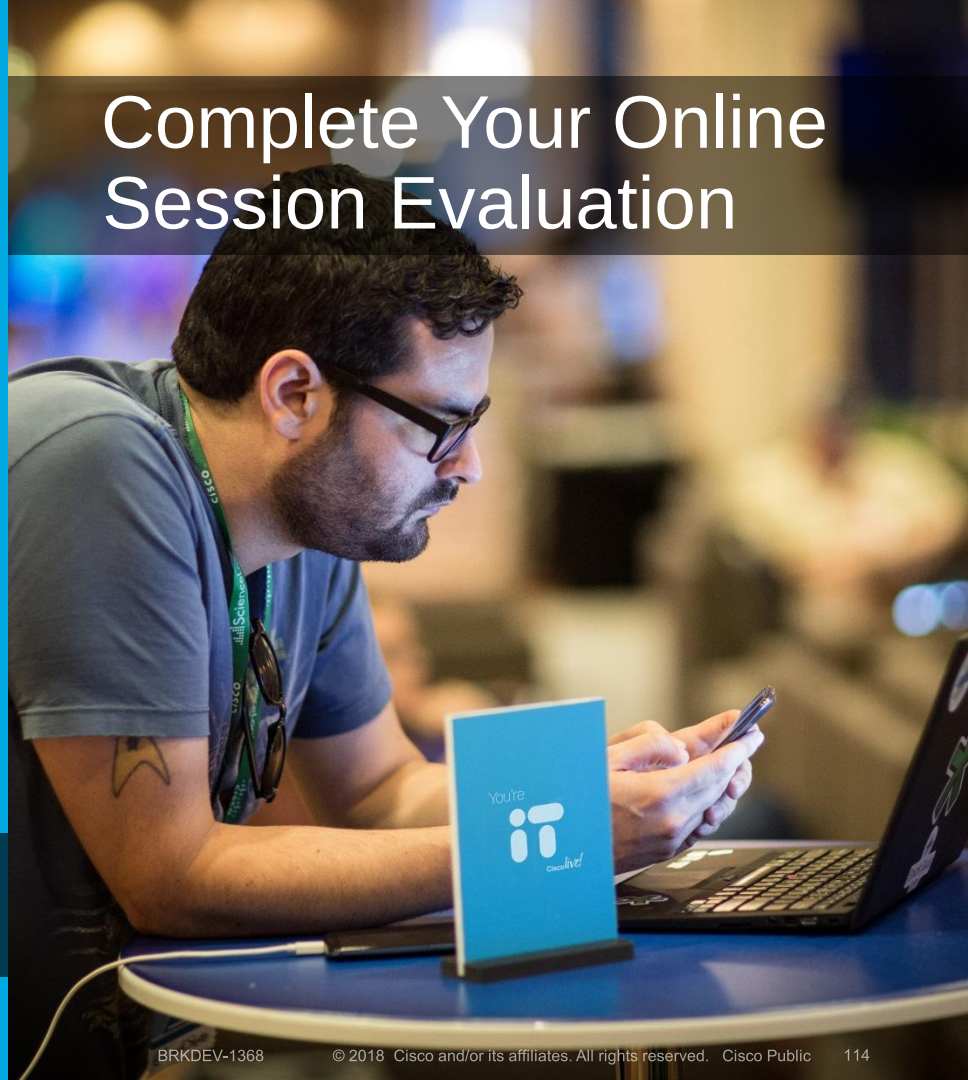


cs.co/ciscolivebot#BRKDEV-1368

Complete Your Online Session Evaluation

- Please complete your Online Session Evaluations after each session
- Complete 4 Session Evaluations & the Overall Conference Evaluation (available from Thursday) to receive your Cisco Live T-shirt
- All surveys can be completed via the Cisco Live Mobile App or the Communication Stations

Don’t forget: Cisco Live sessions will be available for viewing on-demand after the event at www.ciscolive.com/global/on-demand-library/.



Continue Your Education

- Demos in the Cisco campus
- Walk-in Self-Paced Labs
- Tech Circle
- Meet the Engineer 1:1 meetings
- Related sessions

Thank you


You're
Cisco live!