Programmability Configuration Guide for Cisco NCS 540 Series Routers, Cisco IOS XR Release 7.0.x

First Published: 2019-08-30

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
   800 553-NETS (6387)
Fax: 408 527-0883
# CONTENTS

## CHAPTER 1

**New and Changed Feature Information** 1

- New and Changed Programmability Features 1

## CHAPTER 2

**Drive Network Automation Using Programmable YANG Data Models** 3

- YANG Data Model 4
  - Components of a YANG Module 5
  - Structure of YANG Data Model 5
- Access the Data Models 7
- Communication Protocols 8
  - NETCONF Protocol 9
  - gRPC Protocol 9

## CHAPTER 3

**Use NETCONF Protocol to Define Network Operations with Data Models** 11

- NETCONF Operations 12
  - Set Router Clock Using Data Model in a NETCONF Session 16
    - Configure Router Clock 18
    - View the Router Clock 20

## CHAPTER 4

**Use gRPC Protocol to Define Network Operations with Data Models** 21

- gRPC Operations 24
- gRPC Network Management Interface 25
- gRPC Network Operations Interface 25
- Configure Interfaces Using Data Models in a gRPC Session 27
  - Enable gRPC Protocol 29
  - Configure Interfaces 30
  - Verify the Interface State 32
# New and Changed Feature Information

This section lists all the new and changed features for the Programmability Configuration Guide.

- New and Changed Programmability Features, on page 1

## New and Changed Programmability Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Changed in Release</th>
<th>Where Documented</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNOI</td>
<td>gRPC Network Operations Interface (gNOI) defines a set of gRPC-based microservices for executing operational commands on network devices. Extensible Manageability Services (EMS) gNOI is the Cisco IOS XR implementation of gNOI. GNOI supports Reboot, RebootStatus, SetPackage, File Get and File Remove RPCs.</td>
<td>Release 7.0.1</td>
<td>Components to Use Data Models chapter, gRPC Network Operations Interface, on page 25</td>
</tr>
<tr>
<td>Support for oc-lldp model for Event-driven Telemetry</td>
<td>The OpenConfig-Link Layer Discovery Protocol (oc-lldp) model defined by the OC community defines configuration and operational state data for the LLDP protocol.</td>
<td>Release 7.0.1</td>
<td>Components to Use Data Models chapter, YANG Data Model, on page 4 Obtain this data model from Github repository.</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Changed in Release</td>
<td>Where Documented</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-------------------</td>
<td>------------------</td>
</tr>
</tbody>
</table>
| Support for oc-if-aggregate data model | The OpenConfig-Interfaces Aggregate (oc-if-aggregate) model defined by the OC community manages aggregated (bundle, link aggregation (LAG)) interfaces. This model augments the existing oc-interfaces data model. | Release 7.0.1 | Components to Use Data Models chapter  
YANG Data Model, on page 4  
Obtain this data model from Github repository. |
CHAPTER 2

Drive Network Automation Using Programmable YANG Data Models

Typically, a network operation center is a heterogeneous mix of various devices at multiple layers of the network. Such network centers require bulk automated configurations to be accomplished seamlessly. CLIs are widely used for configuring and extracting the operational details of a router. But the general mechanism of CLI scraping is not flexible and optimal. Small changes in the configuration require rewriting scripts multiple times. Bulk configuration changes through CLIs are cumbersome and error-prone. These limitations restrict automation and scale. To overcome these limitations, you need an automated mechanism to manage your network.

Cisco IOS XR supports a programmatic way of configuring and collecting operational data of a network device using data models. They replace the process of manual configuration, which is proprietary, and highly text-based. The data models are written in an industry-defined language and is used to automate configuration task and retrieve operational data across heterogeneous devices in a network. Although configurations using CLIs are easier and human-readable, automating the configuration using model-driven programmability results in scalability.

Model-driven programmability provides a simple, flexible and rich framework for device programmability. This programmability framework provides multiple choices to interface with an IOS XR device in terms of transport, protocol and encoding. These choices are decoupled from the models for greater flexibility.

The following image shows the layers in model-driven programmability:
Figure 1: Model-driven Programmability Layers

Data models provide access to the capabilities of the devices in a network using Network Configuration Protocol (NETCONF Protocol) or google-defined Remote Procedure Calls (gRPC Protocol). The operations on the router are carried out by the protocols using YANG models to automate and programme operations in a network.

Benefits of Data Models

Configuring routers using data models overcomes drawbacks posed by traditional router management because the data models:

- Provide a common model for configuration and operational state data, and perform NETCONF actions.
- Use protocols to communicate with the routers to get, manipulate and delete configurations in a network.
- Automate configuration and operation of multiple routers across the network.

This article describes how you benefit from using data models to programmatically manage your network operations.

- YANG Data Model, on page 4
- Access the Data Models, on page 7
- Communication Protocols, on page 8

YANG Data Model

A YANG module defines a data model through the data of the router, and the hierarchical organization and constraints on that data. Each module is uniquely identified by a namespace URL. The YANG models describe the configuration and operational data, perform actions, remote procedure calls, and notifications for network devices.
The YANG models must be obtained from the router. The models define a valid structure for the data that is exchanged between the router and the client. The models are used by NETCONF and gRPC-enabled applications.

YANG models can be:

- **Cisco-specific models**: For a list of supported models and their representation, see Native models.

- **Common models**: These models are industry-wide standard YANG models from standard bodies, such as IETF and IEEE. These models are also called Open Config (OC) models. Like synthesized models, the OC models have separate YANG models defined for configuration data and operational data, and actions.

For a list of supported OC models and their representation, see OC models.

For more details about YANG, refer RFC 6020 and 6087.

## Components of a YANG Module

A YANG module defines a single data model. However, a module can reference definitions in other modules and sub-modules by using one of these statements:

- **import** imports external modules
- **include** includes one or more sub-modules
- **augment** provides augmentations to another module, and defines the placement of new nodes in the data model hierarchy
- **when** defines conditions under which new nodes are valid
- **prefix** references definitions in an imported module

The YANG models configure a feature, retrieve the operational state of the router, and perform actions.

---

**Note**

The gRPC YANG path or JSON data is based on YANG module name and not YANG namespace.

## Structure of YANG Data Model

Data models handle the following types of requirements on routers (RFC 6244):

- **Configuration data**: A set of writable data that is required to transform a system from an initial default state into its current state. For example, configuring entries of the IP routing tables, configuring the interface MTU to use a specific value, configuring an ethernet interface to run at a given speed, and so on.

- **Operational state data**: A set of data that is obtained by the system at runtime and influences the behavior of the system in a manner similar to configuration data. However, in contrast to configuration data, operational state data is transient. The data is modified by interactions with internal components or other systems using specialized protocols. For example, entries obtained from routing protocols such as OSPF, attributes of the network interfaces, and so on.
- **Actions**: A set of NETCONF actions that support robust network-wide configuration transactions. When a change is attempted that affects multiple devices, the NETCONF actions simplify the management of failure scenarios, resulting in the ability to have transactions that will dependably succeed or fail atomically.

For more information about Data Models, see RFC 6244.

YANG data models can be represented in a hierarchical, tree-based structure with nodes. This representation makes the models easy to understand.

Each feature has a defined YANG model, which is synthesized from schemas. A model in a tree format includes:

- Top level nodes and their subtrees
- Subtrees that augment nodes in other YANG models
- Custom RPCs

YANG defines four node types. Each node has a name. Depending on the node type, the node either defines a value or contains a set of child nodes. The nodes types for data modeling are:

- **leaf node** - contains a single value of a specific type
- **leaf-list node** - contains a sequence of leaf nodes
- **list node** - contains a sequence of leaf-list entries, each of which is uniquely identified by one or more key leaves
- **container node** - contains a grouping of related nodes that have only child nodes, which can be any of the four node types

### Structure of CDP Data Model

Cisco Discovery Protocol (CDP) configuration has an inherent augmented model (interface-configuration). The augmentation indicates that CDP can be configured at both the global configuration level and the interface configuration level. The data model for CDP interface manager in tree structure is:

```
module: Cisco-IOS-XR-cdp-cfg
  %--rw cdp
     %--rw timer? uint32
     %--rw advertise-v1-only? empty
     %--rw enable? boolean
     %--rw hold-time? uint32
     %--rw log-adjacency? empty
  augment /a1:interface-configurations/a1:interface-configuration:
    %--rw cdp
       %--rw enable? empty
```

In the CDP YANG model, the augmentation is expressed as:

```
augment "/a1:interface-configurations/a1:interface-configuration" {
  container cdp {
    description "Interface specific CDP configuration";
    leaf enable {
      type empty;
      description "Enable or disable CDP on an interface";
    }
  }
  description
```
CDP Operational YANG:

The structure of a data model can be explored using a YANG validator tool such as `pyang` and the data model can be formatted in a tree structure. The following example shows the CDP operational model in tree format.

```yamng
module: Cisco-IOS-XR-cdp-oper
  +--ro cdp
    +--ro nodes
      +--ro node* [node-name]
        +--ro neighbors
          | +--ro details
          |   | +--ro detail*
          |   |   | +--ro interface-name? xr:Interface-name
          |   |   | +--ro device-id? string
          |   |   | +--ro cdp-neighbor*
          |   |   |   | +--ro detail
          |   |   |   |   | +--ro network-addresses
          |   |   |   |   |   | +--ro cdp.addr-entry*
          |   |   |   |   |   |   | +--ro address
          |   |   |   |   |   |   |   | +--ro address-type? Cdp-l3-addr-protocol
          |   |   |   |   |   |   |   | +--ro ipv4-address? inet:ipv4-address
          |   |   |   |   |   |   |   | +--ro ipv6-address? In6-addr
          |   |   |   |   | +--ro protocol-hello-list
          |   |   |   |   |   | +--ro cdp-prot-hello-entry*
          |   |   |   |   |   |   | +--ro hello-message? yang:hex-string
          |   |   |   |   |   | +--ro version? string
          |   |   |   |   |   | +--ro vtp-domain? string
          |   |   |   |   |   | +--ro native-vlan? uint32
          |   |   |   |   |   | +--ro duplex? Cdp-duplex
          |   |   |   |   |   | +--ro system-name? string
          |   |   |   |   | +--ro receiving-interface-name? xr:Interface-name
          |   |   |   |   | +--ro device-id? string
          |   |   |   |   | +--ro port-id? string
          |   |   |   |   | +--ro header-version? uint8
          |   |   |   |   | +--ro hold-time? uint16
          |   |   |   |   | +--ro capabilities? string
          |   |   |   |   | +--ro platform? string
```

Access the Data Models

You can access the Cisco IOS XR native and OpenConfig data models from GitHub, a software development platform that provides hosting services for version control.

You can also access the supported data models from the router. The router ships with the YANG files that define the data models. Use NETCONF protocol to view the data models available on the router using `ietf-netconf-monitoring` request.

```xml
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <get>
    <filter type="subtree">
        <schemas/>
    </netconf-state>
```
Communication Protocols

Communication protocols establish connections between the router and the client. The protocols help the client to consume the YANG data models to, in turn, automate and programme network operations.

YANG uses one of these protocols:

- Network Configuration Protocol (NETCONF)
• gRPC (google-defined Remote Procedure Calls)

The transport and encoding mechanisms for these two protocols are shown in the table:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Transport</th>
<th>Encoding/Decoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETCONF</td>
<td>ssh</td>
<td>xml</td>
</tr>
<tr>
<td>gRPC</td>
<td>http/2</td>
<td>json</td>
</tr>
</tbody>
</table>

**NETCONF Protocol**

NETCONF provides mechanisms to install, manipulate, or delete the configuration on network devices. It uses an Extensible Markup Language (XML)-based data encoding for the configuration data, as well as protocol messages. You use a simple NETCONF RPC-based (Remote Procedure Call) mechanism to facilitate communication between a client and a server. To get started with issuing NETCONF RPCs to configure network features using data models, see Use NETCONF Protocol to Define Network Operations with Data Models, on page 11.

**gRPC Protocol**

gRPC is an open-source RPC framework. It is based on Protocol Buffers (Protobuf), which is an open source binary serialization protocol. gRPC provides a flexible, efficient, automated mechanism for serializing structured data, like XML, but is smaller and simpler to use. You define the structure by defining protocol buffer message types in .proto files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. To get started with issuing NETCONF RPCs to configure network features using data models, see Use gRPC Protocol to Define Network Operations with Data Models, on page 21.
gRPC Protocol
Use NETCONF Protocol to Define Network Operations with Data Models

XR devices ship with the YANG files that define the data models they support. Using a management protocol such as NETCONF or gRPC, you can programmatically query a device for the list of models it supports and retrieve the model files.

Network Configuration Protocol (NETCONF) is a standard transport protocol that communicates with network devices. NETCONF provides mechanisms to edit configuration data and retrieve operational data from network devices. The configuration data represents the way interfaces, routing protocols and other network features are provisioned. The operational data represents the interface statistics, memory utilization, errors, and so on.

NETCONF uses an Extensible Markup Language (XML)-based data encoding for the configuration data, as well as protocol messages. It uses a simple RPC-based (Remote Procedure Call) mechanism to facilitate communication between a client and a server. The client can be a script or application that runs as part of a network manager. The server is a network device such as a router. NETCONF defines how to communicate with the devices, but does not handle what data is exchanged between the client and the server.

To enable NETCONF, use the `ssh server capability netconf-xml` command to reach XML subsystem on port 22.

**NETCONF Session**

A NETCONF session is the logical connection between a network configuration application (client) and a network device (router). The configuration attributes can be changed during any authorized session; the effects are visible in all sessions. NETCONF is connection-oriented, with SSH as the underlying transport. NETCONF sessions are established with a "hello" message, where features and capabilities are announced. Sessions are terminated using `close` or `kill` messages.

**NETCONF Layers**

NETCONF protocol can be partitioned into four layers:
Figure 2: NETCONF Layers

- **Content layer**: includes configuration and notification data
- **Operations layer**: defines a set of base protocol operations invoked as RPC methods with XML-encoded parameters
- **Messages layer**: provides a simple, transport-independent framing mechanism for encoding RPCs and notifications
- **Secure Transport layer**: provides a communication path between the client and the server

For more information about NETCONF, refer RFC 6241.

This article describes, with a use case to configure the local time on a router, how data models help in a faster programmatic configuration as compared to CLI.

- NETCONF Operations, on page 12
- Set Router Clock Using Data Model in a NETCONF Session, on page 16

**NETCONF Operations**

NETCONF defines one or more configuration datastores and allows configuration operations on the datastores. A configuration datastore is a complete set of configuration data that is required to get a device from its initial default state into a desired operational state. The configuration datastore does not include state data or executive commands.

The base protocol includes the following NETCONF operations:

```plaintext
| +++-get-config
| +++-edit-config
| | +++-merge
| | +++-replace
| | +++-create
| | +++-delete
| | +++-remove
| | +++-default-operations
| | | +++-merge
| | | +++-replace
| | | +++-none
| | +++-get
| | +++-lock
| | +++-unlock
| | +++-close-session
| | +++-kill-session
```
These NETCONF operations are described in the following table:

<table>
<thead>
<tr>
<th>NETCONF Operation</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;get-config&gt;</td>
<td>Retrieves all or part of a specified configuration from a named data store</td>
<td>Retrieve specific interface configuration details from running configuration using filter option</td>
</tr>
</tbody>
</table>

```
<rpc message-id="101"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<get-config>
<source>
<running/>
</source>
<filter>
<interface-configurations
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ifmgr-cfg">
<interface-configuration>
<active>act</active>
<interface-name>TenGigE0/0/0/2</interface-name>
</interface-configuration>
</interface-configurations>
</filter>
</get-config>
</rpc>
```

| <get> | Retrieves running configuration and device state information | Retrieve all acl configuration and device state information. |

```
Request:
<get>
<filter>
<ipv4-acl-and-prefix-list
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-acl-oper"/>
</filter>
</get>
```
<table>
<thead>
<tr>
<th>NETCONF Operation</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;edit-config&gt;</td>
<td>Loads all or part of a specified configuration to the specified target configuration</td>
<td>Configure ACL configs using <strong>Merge</strong> operation</td>
</tr>
</tbody>
</table>

```
<rpc message-id="101"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<edit-config>
<target><candidate/></target>
<config
xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
<ipv4-acl-and-prefix-list
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-acl-cfg"
xc:operation="merge">
<accesses>
<access>
<access-list-name>aclv4-1</access-list-name>
<access-list-entries>
<access-list-entry>
<sequence-number>10</sequence-number>
<remark>GUEST</remark>
</access-list-entry>
<access-list-entry>
<sequence-number>20</sequence-number>
<grant>permit</grant>
<source-network>
<source-address>172.0.0.0</source-address>
<source-wild-card-bits>0.0.255.255</source-wild-card-bits>
</source-network>
</access-list-entries>
</access>
</accesses>
</ipv4-acl-and-prefix-list>
</config>
</edit-config>
</rpc>
```

Commit:
```
<rpc message-id="101"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<commit/>
</rpc>
```

<table>
<thead>
<tr>
<th>&lt;lock&gt;</th>
<th>Allows the client to lock the entire configuration datastore system of a device</th>
<th>Lock the running configuration.</th>
</tr>
</thead>
</table>
| Request:           | <rpc message-id="101"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<lock>
<target>
<running/>
</target>
</lock>
</rpc> | Response :<rpc-reply message-id="101"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<ok/>
</rpc-reply> |
### NETCONF Operations

<table>
<thead>
<tr>
<th>NETCONF Operation</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| <Unlock>          | Releases a previously locked configuration. An <unlock> operation will not succeed if either of the following conditions is true: • The specified lock is not currently active. • The session issuing the <unlock> operation is not the same session that obtained the lock. | Lock and unlock the running configuration from the same session. Request:  
```xml
<rpc message-id="101"
 xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <unlock>
   <target>
     <running/>
   </target>
 </unlock>
</rpc>
```
Response -  
```xml
<rpc-reply message-id="101"
 xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <ok/>
</rpc-reply>
```

| <close-session> | Closes the session. The server releases any locks and resources associated with the session and closes any associated connections. | Close a NETCONF session. Request:  
```xml
<rpc message-id="101"
 xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <close-session/>
</rpc>
```
Response:  
```xml
<rpc-reply message-id="101"
 xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <ok/>
</rpc-reply>
```

| <kill-session>  | Aborts operations currently in process, releases locks and resources associated with the session, and close any associated connections. | Abort a session if the ID is other session ID. Request:  
```xml
<rpc message-id="101"
 xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <kill-session>
   <session-id>4</session-id>
 </kill-session>
</rpc>
```
Response:  
```xml
<rpc-reply message-id="101"
 xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <ok/>
</rpc-reply>
```

### NETCONF Operation to Get Configuration

This example shows how a NETCONF <get-config> request works for CDP feature. The client initiates a message to get the current configuration of CDP running on the router. The router responds with the current CDP configuration.
The `<rpc>` element in the request and response messages enclose a NETCONF request sent between the client and the router. The `message-id` attribute in the `<rpc>` element is mandatory. This attribute is a string chosen by the sender and encodes an integer. The receiver of the `<rpc>` element does not decode or interpret this string but simply saves it to be used in the `<rpc-reply>` message. The sender must ensure that the `message-id` value is normalized. When the client receives information from the server, the `<rpc-reply>` message contains the same `message-id`.

**Set Router Clock Using Data Model in a NETCONF Session**

**NETCONF Protocol** is an XML-based protocol used over Secure Shell (SSH) transport to configure a network. The client applications use this protocol to request information from the router, and make configuration changes to the router.

The process for using data models involves:

- Obtain the data models.
- Establish a connection between the router and the client using NETCONF communication protocol.
- Manage the configuration of the router from the client using data models.

**Note**

Configure AAA authorization to restrict users from uncontrolled access. If AAA authorization is not configured, the command and data rules associated to the groups that are assigned to the user are bypassed. An IOS-XR user can have full read-write access to the IOS-XR configuration through Network Configuration Protocol (NETCONF), google-defined Remote Procedure Calls (gRPC) or any YANG-based agents. In order to avoid granting uncontrolled access, enable AAA authorization before setting up any configuration.

The following image shows the tasks involved in using data models.
In this section, you use native data models to configure the router clock and verify the clock state using a NETCONF session.

Consider a network topology with four routers and one controller. The network consists of label edge routers (LER) and label switching routers (LSR). Two routers LER1 and LER2 are label edge routers, and two routers LSR1 and LSR2 are label switching routers. A host is the controller with a gRPC client. The controller communicates with all routers through an out-of-band network. All routers except LER1 are pre-configured with proper IP addressing and routing behavior. Interfaces between routers have a point-to-point configuration with /31 addressing. Loopback prefixes use the format 172.16.255.x/32.

The following image illustrates the network topology:
You use Cisco IOS XR native models `Cisco-IOS-XR-infra-clock-linux-cfg.yang` and `Cisco-IOS-XR-shellutil-oper` to programmatically configure the router clock. You can explore the structure of the data model using YANG validator tools such as `pyang`.

**Before you begin**

Retrieve the list of YANG modules on the router using NETCONF monitoring RPC. For more information, see Access the Data Models, on page 7.

## Configure Router Clock

### Step 1

Explore the native configuration model for the system local time zone.

**Example:**

```
controller:netconf$ pyang --format tree Cisco-IOS-XR-infra-clock-linux-cfg.yang
module: Cisco-IOS-XR-infra-clock-linux-cfg
  +--rw clock
    +--rw time-zone!
    +--rw time-zone-name string
    +--rw area-name string
```

### Step 2

Explore the native operational state model for the system time.

**Example:**

```
controller:netconf$ pyang --format tree Cisco-IOS-XR-shellutil-oper.yang
module: Cisco-IOS-XR-shellutil-oper
  +--ro system-time
    +--ro clock
      | +--ro year? uint16
      | +--ro month? uint8
      | +--ro day? uint8
```
Step 3  Retrieve the current time on router LER1.

Example:

controller:netconf$ more xr-system-time-oper.xml <system-time
controller:netconf$ netconf get --filter xr-system-time-oper.xml
198.18.1.11:830
<?xml version="1.0" ?>
<system-time xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-shellutil-oper">
  <clock>
    <year>2019</year>
    <month>8</month>
    <day>22</day>
    <hour>17</hour>
    <minute>30</minute>
    <second>37</second>
    <millisecond>690</millisecond>
    <wday>1</wday>
    <time-zone>UTC</time-zone>
    <time-source>calendar</time-source>
  </clock>
  <uptime>
    <host-name>ler1</host-name>
    <uptime>851237</uptime>
  </uptime>
</system-time>

Notice that the timezone UTC indicates that a local timezone is not set.

Step 4  Configure Pacific Standard Time (PST) as local time zone on LER1.

Example:

controller:netconf$ more xr-system-time-oper.xml <system-time
controller:netconf$ netconf get --filter xr-system-time-oper.xml
<username>:<password>@198.18.1.11:830
<?xml version="1.0" ?>
<system-time xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-shellutil-oper">
  <clock>
    <year>2019</year>
    <month>8</month>
    <day>22</day>
    <hour>9</hour>
    <minute>52</minute>
    <second>10</second>
    <millisecond>134</millisecond>
    <wday>1</wday>
    <time-zone>PST</time-zone>
    <time-source>calendar</time-source>
  </clock>
</system-time>
View the Router Clock

Verify that the router clock is set to PST time zone.

```
controller:netconf$ more xr-system-time-oper.xml
<system-time xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-shellutil-oper"/>

controller:netconf$ netconf get --filter xr-system-time-oper.xml
<username>::<password>@198.18.1.11:830
<?xml version="1.0" ?>
<system-time xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-shellutil-oper">
    <clock>
        <year>2018</year>
        <month>12</month>
        <day>22</day>
        <hour>9</hour>
        <minute>52</minute>
        <second>10</second>
        <millisecond>134</millisecond>
        <wday>1</wday>
        <time-zone>PST</time-zone>
        <time-source>calendar</time-source>
    </clock>
    <uptime>
        <host-name>ler1</host-name>
        <uptime>852530</uptime>
    </uptime>
</system-time>
```

In summary, router LER1, which had no local timezone configuration, is programmatically configured using data models.
CHAPTER 4

Use gRPC Protocol to Define Network Operations with Data Models

XR devices ship with the YANG files that define the data models they support. Using a management protocol such as NETCONF or gRPC, you can programmatically query a device for the list of models it supports and retrieve the model files.

gRPC is an open-source RPC framework. It is based on Protocol Buffers (Protobuf), which is an open source binary serialization protocol. gRPC provides a flexible, efficient, automated mechanism for serializing structured data, like XML, but is smaller and simpler to use. You define the structure using protocol buffer message types in .proto files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs.

gRPC encodes requests and responses in binary. gRPC is extensible to other content types along with Protobuf. The Protobuf binary data object in gRPC is transported over HTTP/2.

gRPC supports distributed applications and services between a client and server. gRPC provides the infrastructure to build a device management service to exchange configuration and operational data between a client and a server. The structure of the data is defined by YANG models.

Cisco gRPC IDL uses the protocol buffers interface definition language (IDL) to define service methods, and define parameters and return types as protocol buffer message types. The gRPC requests are encoded and sent to the router using JSON. Clients can invoke the RPC calls defined in the IDL to program the router.

The following example shows the syntax of the proto file for a gRPC configuration:

```
syntax = "proto3";
package IOSXRExtensibleManagabilityService;

service gRPCConfigOper {
    rpc GetConfig(ConfigGetArgs) returns(stream ConfigGetReply) {};
    rpc MergeConfig(ConfigArgs) returns(ConfigReply) {};
    rpc DeleteConfig(ConfigArgs) returns(ConfigReply) {};
    rpc ReplaceConfig(ConfigArgs) returns(ConfigReply) {};
    rpc CliConfig(CliConfigArgs) returns(CliConfigReply) {};
    rpc GetOper(GetOperArgs) returns(stream GetOperReply) {};
    rpc CommitReplace(CommitReplaceArgs) returns(CommitReplaceReply) {};
```
Example for gRPCExec configuration:

```protobuf
def message ConfigGetArgs {
  int64 ReqId = 1;
  string yangpathjson = 2;
}

def message ConfigGetReply {
  int64 ResReqId = 1;
  string yangjson = 2;
  string errors = 3;
}

def message GetOperArgs {
  int64 ReqId = 1;
  string yangpathjson = 2;
}

def message GetOperReply {
  int64 ResReqId = 1;
  string yangjson = 2;
  string errors = 3;
}

def message ConfigArgs {
  int64 ReqId = 1;
  string yangjson = 2;
}

def message ConfigReply {
  int64 ResReqId = 1;
  string errors = 2;
}

def message CliConfigArgs {
  int64 ReqId = 1;
  string cli = 2;
}

def message CliConfigReply {
  int64 ResReqId = 1;
  string errors = 2;
}

def message CommitReplaceArgs {
  int64 ReqId = 1;
  string cli = 2;
  string yangjson = 3;
}

def message CommitReplaceReply {
  int64 ResReqId = 1;
  string errors = 2;
}
```

```protobuf
def service gRPCExec {
  rpc ShowCmdTextOutput(ShowCmdArgs) returns(stream ShowCmdTextReply) {};
  rpc ShowCmdJSONOutput(ShowCmdArgs) returns(stream ShowCmdJSONReply) {};
  rpc ActionJSON(ActionJSONArgs) returns(stream ActionJSONReply) {};
}

def message ShowCmdArgs {
```
int64 ReqId = 1;
string cli = 2;
}

message ShowCmdTextReply {
int64 ResReqId = 1;
string output = 2;
string errors = 3;
}

message ActionJSONArgs {
int64 ReqId = 1;
string yangpathjson = 2;
}

message ActionJSONReply {
int64 ResReqId = 1;
string yangjson = 2;
string errors = 3;
}

Example for OpenConfiggRPC configuration:

service OpenConfiggRPC {
  rpc SubscribeTelemetry(SubscribeRequest) returns (stream SubscribeResponse) {};
  rpc UnSubscribeTelemetry(CancelSubscribeReq) returns (SubscribeResponse) {};
  rpc GetModels(GetModelsInput) returns (GetModelsOutput) {};
}

message GetModelsInput {
  uint64 requestId = 1;
  string name = 2;
  string namespace = 3;
  string version = 4;
  enum MODLE_REQUEST_TYPE {
    SUMMARY = 0;
    DETAIL = 1;
  }
  MODLE_REQUEST_TYPE requestType = 5;
}

message GetModelsOutput {
  uint64 requestId = 1;
  message ModelInfo {
    string name = 1;
    string namespace = 2;
    string version = 3;
    GET_MODEL_TYPE modelType = 4;
    string modelData = 5;
  }
  repeated ModelInfo models = 2;
  OC_RPC_RESPONSE_TYPE responseCode = 3;
  string msg = 4;
}

This article describes, with a use case to configure interfaces on a router, how data models helps in a faster programmatic and standards-based configuration of a network, as compared to CLI.

- gRPC Operations, on page 24
- gRPC Network Management Interface, on page 25
- gRPC Network Operations Interface, on page 25
- Configure Interfaces Using Data Models in a gRPC Session, on page 27
gRPC Operations

You can issue the following gRPC operations:

<table>
<thead>
<tr>
<th>gRPC Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetConfig</td>
<td>Retrieves a configuration</td>
</tr>
<tr>
<td>GetModels</td>
<td>Gets the supported Yang models on the router</td>
</tr>
<tr>
<td>MergeConfig</td>
<td>Appends to an existing configuration</td>
</tr>
<tr>
<td>DeleteConfig</td>
<td>Deletes a configuration</td>
</tr>
<tr>
<td>ReplaceConfig</td>
<td>Modifies a part of an existing configuration</td>
</tr>
<tr>
<td>CommitReplace</td>
<td>Replaces existing configuration with the new configuration file provided</td>
</tr>
<tr>
<td>GetOper</td>
<td>Gets operational data using JSON</td>
</tr>
<tr>
<td>CliConfig</td>
<td>Invokes the CLI configuration</td>
</tr>
<tr>
<td>ShowCmdTextOutput</td>
<td>Displays the output of show command</td>
</tr>
<tr>
<td>ShowCmdJSONOutput</td>
<td>Displays the JSON output of show command</td>
</tr>
</tbody>
</table>

**gRPC Operation to Get Configuration**

This example shows how a gRPC GetConfig request works for CDP feature.

The client initiates a message to get the current configuration of CDP running on the router. The router responds with the current CDP configuration.

```
grpc GetConfig
{
  "Cisco-IOS-XR-cdp-cfg:cdp": {
    "cdp": "running-configuration"
  }
}
```

```
{
  "Cisco-IOS-XR-cdp-cfg:cdp": {
    "timer": 50,
    "enable": true,
    "log-adjacency": [
      null
    ],
    "hold-time": 180,
    "advertise-v1-only": [
      null
    ]
  }
}
```
gRPC Network Management Interface

gRPC Network Management Interface (gNMI) is a gRPC-based network management protocol used to modify, install or delete configuration from network devices. It is also used to view operational data, control and generate telemetry streams from a target device to a data collection system. It uses a single protocol to manage configurations and stream telemetry data from network devices.

gNMI uses gRPC as the transport protocol and the configuration is same as that of gRPC. These gNMI RPCs are supported:

<table>
<thead>
<tr>
<th>gNMI RPC</th>
<th>gNMI RPC Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capabilities</td>
<td></td>
<td>Initial handshake between the network device (server) and the client to exchange capability information such as supported data models</td>
</tr>
<tr>
<td>Set</td>
<td>message SetRequest {}</td>
<td>Modifies data associated with a model on a network device from a client</td>
</tr>
<tr>
<td>Get</td>
<td>message GetRequest {}</td>
<td>Retrieves data from a network device</td>
</tr>
<tr>
<td>Subscribe</td>
<td>message SubscribeRequest {}</td>
<td>Control data subscriptions on server</td>
</tr>
</tbody>
</table>

For more information about gNMI, see Github.

---

gRPC Network Operations Interface

G NOI uses gRPC as the transport protocol and the configuration is same as that of gRPC. These gNOI RPCs are supported:

**Get RPC**

Streams the contents of a file from the target.

```
RPC to 10.105.57.106:57900
RPC start time: 20:58:27.513638
---------------------File Get Request---------------------
RPC start time: 20:58:27.513668
remote_file: "harddisk:/giso_image_repo/test.log"

---------------------File Get Response---------------------
RPC end time: 20:58:27.518413
contents: "GNOI 

hash {}"
```
Remove RPC
Remove the specified file from the target.

RPC to 10.105.57.106:57900
RPC start time: 21:07:57.089554
---------------------File Remove Request---------------------
remote_file: "harddisk:/sample.txt"
---------------------File Remove Response---------------------
RPC end time: 21:09:27.796217
File removal harddisk:/sample.txt successful

Reboot RPC
Reloads a requested target.

RPC to 10.105.57.106:57900
RPC start time: 21:12:49.811536
---------------------Reboot Request---------------------
RPC start time: 21:12:49.811561
method: COLD
message: "Test Reboot"
subcomponents {
  origin: "openconfig-platform"
  elem {
    name: "components"
  }
  elem {
    name: "component"
    key {
      key: "name"
      value: "0/RP0"
    }
    elem {
      name: "state"
    }
    elem {
      name: "location"
    }
  }
}---------------------Reboot Request---------------------
RPC end time: 21:12:50.023604

Set Package RPC
Places software package on the target.

RPC to 10.105.57.106:57900
RPC start time: 21:12:49.811536
---------------------Set Package Request---------------------
RPC start time: 15:33:34.378745
Sending SetPackage RPC
package {
  filename: "harddisk:/giso_image_repo/<platform-version>-giso.iso"
  activate: true
}
method: MD5
Reboot Status RPC

Returns the status of reboot for the target.

RPC to 10.105.57.106:57900
RPC start time: 22:27:34.209473
---------------------Reboot Status Request---------------------
subcomponents {
    origin: "openconfig-platform"
    elem {
        name: "components"
    }
    elem {
        name: "component"
        key {
            key: "name"
            value: "0/RP0"
        }
    }
    elem {
        name: "state"
    }
    elem {
        name: "location"
    }
}
---------------------Reboot Status Response---------------------

Active : False
Wait : 0
When : 0
Reason : Test Reboot
Count : 0

To send gNOI RPC requests, user needs a client that implements the gNOI client interface for each RPC.

All messages within the gRPC service definition are defined as protocol buffers (proto files). gNOI OpenConfig proto files are located in Github.

Configure Interfaces Using Data Models in a gRPC Session

Google-defined remote procedure call (gRPC Protocol) is an open-source RPC framework. gRPC supports IPv4 and IPv6 address families. The client applications use this protocol to request information from the router, and make configuration changes to the router.

The process for using data models involves:

• Obtain the data models.

• Establish a connection between the router and the client using gRPC communication protocol.

• Manage the configuration of the router from the client using data models.
Configure AAA authorization to restrict users from uncontrolled access. If AAA authorization is not configured, the command and data rules associated to the groups that are assigned to the user are bypassed. An IOS-XR user can have full read-write access to the IOS-XR configuration through Network Configuration Protocol (NETCONF), google-defined Remote Procedure Calls (gRPC) or any YANG-based agents. In order to avoid granting uncontrolled access, enable AAA authorization before setting up any configuration.

In this section, you use native data models to configure loopback and ethernet interfaces on a router using a gRPC session.

Consider a network topology with four routers and one controller. The network consists of label edge routers (LER) and label switching routers (LSR). Two routers LER1 and LER2 are label edge routers, and two routers LSR1 and LSR2 are label switching routers. A host is the controller with a gRPC client. The controller communicates with all routers through an out-of-band network. All routers except LER1 are pre-configured with proper IP addressing and routing behavior. Interfaces between routers have a point-to-point configuration with /31 addressing. Loopback prefixes use the format 172.16.255.x/32.

The following image illustrates the network topology:

*Figure 5: Network Topology for gRPC session*

You use Cisco IOS XR native model Cisco-IOS-XR-ifmgr-cfg.yang to programmatically configure router LER1.

**Before you begin**

- Retrieve the list of YANG modules on the router using NETCONF monitoring RPC. For more information, see Access the Data Models, on page 7.
• Configure Transport Layer Security (TLS). Enabling gRPC protocol uses the default HTTP/2 transport with no TLS. gRPC mandates AAA authentication and authorization for all gRPC requests. If TLS is not configured, the authentication credentials are transferred over the network unencrypted. Enabling TLS ensures that the credentials are secure and encrypted. Non-TLS mode can only be used in secure internal network.

**Enable gRPC Protocol**

To configure network devices and view operational data, gRPC protocol must be enabled on the server. In this example, you enable gRPC protocol on LER1, the server.

### Step 1
Enable gRPC over an HTTP/2 connection.

**Example:**

```plaintext
Router#configure
Router(config)#grpc
Router(config-grpc)#port <port-number>
```

The port number ranges from 57344 to 57999. If a port number is unavailable, an error is displayed.

### Step 2
Set the session parameters.

**Example:**

```plaintext
Router(config)#grpc{ address-family | dscp | max-request-per-user | max-request-total | max-streams | max-streams-per-user | no-tls | service-layer | tls-cipher | tls-mutual | tls-trustpoint | vrf }
```

where:

- **address-family**: set the address family identifier type
- **dscp**: set QoS marking DSCP on transmitted gRPC
- **max-request-per-user**: set the maximum concurrent requests per user
- **max-request-total**: set the maximum concurrent requests in total
- **max-streams**: set the maximum number of concurrent gRPC requests. The maximum subscription limit is 128 requests. The default is 32 requests
- **max-streams-per-user**: set the maximum concurrent gRPC requests for each user. The maximum subscription limit is 128 requests. The default is 32 requests
- **no-tls**: disable transport layer security (TLS). The TLS is enabled by default.
- **service-layer**: enable the grpc service layer configuration
- **tls-cipher**: enable the gRPC TLS cipher suites
- **tls-mutual**: set the mutual authentication
- **tls-trustpoint**: configure trustpoint
- **server-vrf**: enable server vrf
After gRPC is enabled, use the YANG data models to manage network configurations.

**Configure Interfaces**

In this example, you configure interfaces using Cisco IOS XR native model `Cisco-IOS-XR-ifmgr-cfg.yang`. You gain an understanding about the various gRPC operations while you configure the interface. For the complete list of operations, see [gRPC Operations](#), on page 24. In this example, you merge configurations with `merge-config` RPC, retrieve operational statistics using `get-oper` RPC, and delete a configuration using `delete-config` RPC. You can explore the structure of the data model using YANG validator tools such as `pyang`.

LER1 is the gRPC server, and a command line utility `grpc` is used as a client on the controller. This utility does not support YANG and, therefore, does not validate the data model. The server, LER1, validates the data mode.

**Step 1**

Explore the XR configuration model for interfaces and its IPv4 augmentation.

**Example:**

```
controller:grpc$ pyang --format tree --tree-depth 3 Cisco-IOS-XR-ifmgr-cfg.yang
Cisco-IOS-XR-ipv4-io-cfg.yang
module: Cisco-IOS-XR-ifmgr-cfg
  | +--rw global-interface-configuration
  |   | +--rw link-status? Link-status-enum
  |   | +--rw interface-configurations
  |   |   | +--rw interface-configuration* [active interface-name]
  |   |   |   | +--rw dampening
  |   |   |   |   | ...   +--rw mtus
  |   |   |   |   | | ...   +--rw encapsulation
  |   |   |   |   | | ...   +--rw shutdown? empty
  |   |   |   |   | +--rw interface-virtual? empty
  |   |   |   |   | +--rw secondary-admin-state? Secondary-admin-state-enum
  |   |   |   |   | +--rw interface-mode-non-physical? Interface-mode-enum
  |   |   |   |   | +--rw bandwidth? uint32
  |   |   |   |   | +--rw link-status? empty
  |   |   |   |   | +--rw description? string
  |   |   |   |   | +--rw active Interface-active
  |   |   |   |   | +--rw interface-name xr:Interface-name
  |   |   |   |   | +--rw ipv4-io-cfg:ipv4-network
  |   |   |   |   | | ...   +--rw ipv4-io-cfg:ipv4-network-forwarding ...
```

**Step 2**

Configure a loopback0 interface on LER1.

a) Configure loopback interface Loopback0 and assign an IP address.

**Example:**

```
controller:grpc$ more xr-interfaces-lo0-cfg.json
{
  "Cisco-IOS-XR-ifmgr-cfg:interface-configurations":
    { "interface-configuration": [
      {
        "active": "act",
      
```
"interface-name": "Loopback0",
"description": "LOCAL TERMINATION ADDRESS",
"interface-virtual": [
  null
],
"Cisco-IOS-XR-ipv4-io-cfg:ipv4-network": {
  "addresses": {
    "primary": {
      "address": "172.16.255.1",
      "netmask": "255.255.255.255"
    }
  }
}
}
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
]
Step 4

Enable the ethernet interface GigabitEthernet 0/0/0/0 on LER1 to bring up the interface. To do this, delete shutdown configuration for the interface.

Example:

```
ccontroller:grpc$ grpcc -username admin -password admin -oper delete-config
-server_addr 198.18.1.11:57400 -yang_path "${< xr-interfaces-gi0-shutdown-cfg.json }"
```

emsDeleteConfig: Sending ReqId 1, yangJson {
"Cisco-IOS-XR-ifmgr-cfg:interface-configurations": {
  "interface-configuration": [
    {
      "active": "act",
      "interface-name": "GigabitEthernet0/0/0/0",
      "shutdown": [
        null
      ]
    }
  ]
}
}
emsDeleteConfig: Received ReqId 1, Response ' '

Verify the Interface State

Verify that the loopback interface and the ethernet interface on router LER1 are operational.

```
ccontroller:grpc$ grpcc -username admin -password admin -oper get-oper
-server_addr 198.18.1.11:57400 -oper_yang_path "${< xr-interfaces-briefs-oper-filter.json }"
```

emsGetOper: Sending ReqId 1, yangPath {
"Cisco-IOS-XR-pfi-im-cmd-oper:interfaces": {
  "interface-briefs": [
    null
  ]
}
}
"Cisco-IOS-XR-pfi-im-cmd-oper:interfaces": {
  "interface-briefs": {
    "interface-brief": {
      "interface-name": "GigabitEthernet0/0/0/0",
      "interface": "GigabitEthernet0/0/0/0",
      "type": "IFT_GETHERNET",
      "state": "im-state-up",
      "actual-state": "im-state-up",
      "line-state": "im-state-up",
      "actual-line-state": "im-state-up",
      "encapsulation": "ether",
      "encapsulation-type-string": "ARPA",
      "mtu": 1514,
      "sub-interface-mtu-overhead": 0,
      "l2-transport": false,
      "bandwidth": 1000000
    }
  ]
}
"interface-name": "GigabitEthernet0/0/0/1",
"interface": "GigabitEthernet0/0/0/1",
"type": "IFT_GETHERNET",
"state": "im-state-up",
"actual-state": "im-state-up",
"line-state": "im-state-up",
"actual-line-state": "im-state-up",
"encapsulation": "ether",
"encapsulation-type-string": "ARPA",
"mtu": 1514,
"sub-interface-mtu-overhead": 0,
"l2-transport": false,
"bandwidth": 1000000
},
{
"interface-name": "Loopback0",
"interface": "Loopback0",
"type": "IFT_LOOPBACK",
"state": "im-state-up",
"actual-state": "im-state-up",
"line-state": "im-state-up",
"actual-line-state": "im-state-up",
"encapsulation": "loopback",
"encapsulation-type-string": "Loopback",
"mtu": 1500,
"sub-interface-mtu-overhead": 0,
"l2-transport": false,
"bandwidth": 0
},
{
"interface-name": "MgmtEth0/RP0/CPU0/0",
"interface": "MgmtEth0/RP0/CPU0/0",
"type": "IFT_ETHERNET",
"state": "im-state-up",
"actual-state": "im-state-up",
"line-state": "im-state-up",
"actual-line-state": "im-state-up",
"encapsulation": "ether",
"encapsulation-type-string": "ARPA",
"mtu": 1514,
"sub-interface-mtu-overhead": 0,
"l2-transport": false,
"bandwidth": 1000000
},
{
"interface-name": "Null0",
"interface": "Null0",
"type": "IFT_NULL",
"state": "im-state-up",
"actual-state": "im-state-up",
"line-state": "im-state-up",
"actual-line-state": "im-state-up",
"encapsulation": "null",
"encapsulation-type-string": "Null",
"mtu": 1500,
"sub-interface-mtu-overhead": 0,
"l2-transport": false,
"bandwidth": 0
}

emsGetOper: ReqId 1, byteRecv: 2325
In summary, router LER1, which had minimal configuration, is now programatically configured using data models with an ethernet interface and is assigned a loopback address. Both these interfaces are operational and ready for network provisioning operations.