

gNMI Protocol

The gNMI Protocol feature describes the model-driven configuration and retrieval of operational data using the gRPC Network Management Interface (gNMI) capabilities, and the Get, Set, and Subscribe remote procedure calls (RPCs). gNMI Version 0.4.0 is supported.

- Restrictions for gNMI Protocol, on page 1
- Information About the gNMI Protocol, on page 2
- How to Enable the gNMI Protocol, on page 15
- Configuration Examples for the gNMI Protocol, on page 21
- Additional References for the gNMI Protocol, on page 21
- Feature Information for the gNMI Protocol, on page 22

Restrictions for gNMI Protocol

The following restrictions apply to the gNMI Protocol feature:

- BYTES and ASCII encoding options are not supported.

  PROTO encoding is supported from Cisco IOS XE Dublin 17.11.1.

- JSON IETF keys must contain a YANG prefix where the namespace of the child elements differs from that of the parent. This means that the routed VLAN derived from augmentation in openconfig-vlan.yang must be entered as *oc-vlan:routed-vlan* because it is different from the namespace of the parent nodes (parent nodes have the prefix oc-if)

- GetRequest:

  - Operational data filtering is not supported.

  - Use models are not supported. These are a set of model data messages indicating the schema definition modules that define the data elements that must be returned in response to a Get RPC call.

- GetResponse:

  - Alias, which is a string that provides an alternate name for a prefix specified within the GetResponse notification message is not supported.

  - Delete, which is a set of paths that are to be removed from a data tree is not supported.

Information About the gNMI Protocol

About GNMI

gNMI is gRPC Network Management Interface developed by Google. gNMI provides the mechanism to install, manipulate, and delete the configuration of network devices, and also to view operational data. The content provided through gNMI can be modeled using YANG.

gRPC is a remote procedure call developed by Google for low-latency, scalable distributions with mobile clients communicating to a cloud server. gRPC carries gNMI, and provides the means to formulate and transmit data and operation requests.

When a gNMI service failure occurs, the gNMI broker (GNMIB) will indicate an operational change of state from up to down, and all RPCs will return a service unavailable message until the database is up and running. Upon recovery, the GNMIB will indicate a change of operation state from down to up, and resume normal handling of RPCs.

gNMI supports <subscribe> RPC services. For more information, see the Model-Driven Telemetry chapter.

JSON IETF Encoding for YANG Data Trees

RFC 7951 defines JavaScript Object Notation (JSON) encoding for YANG data trees and their subtrees. gNMI uses JSON for encoding data in its content layer.

The JSON type indicates that the value is encoded as a JSON string. JSON_IETF-encoded data must conform to the rules for JSON serialisation described in RFC 7951. Both the client and target must support JSON encoding.

Instances of YANG data nodes (leafs, containers, leaf-lists, lists, anydata nodes, and anyxml nodes) are encoded as members of a JSON object or name/value pairs. Encoding rules are identical for all types of data trees, such as configuration data, state data, parameters of RPC operations, actions, and notifications.

Every data node instance is encoded as a name/value pair where the name is formed from the data node identifier. The value depends on the category of the data node.

The leaf Data Node

A leaf node has a value, but no children, in a data tree. A leaf instance is encoded as a name/value pair. This value can be a string, number, literal true or false, or the special array [null], depending on the type of the leaf. In the case that the data item at the specified path is a leaf node (which means it has no children, and an associated value) the value of that leaf is encoded directly. (A bare JSON value is included; it does not require a JSON object.)

The following example shows a leaf node definition:

```
leaf foo {
 type uint8;
}
```

The following is a valid JSON-encoded instance:

```
"foo": 123
```

Proto Encoding

gNMI protocol supports PROTO encoding along with the already supported JSON and JSON_IETF formats. The *gnmi.proto* file represents the blueprint for generating a complete set of client and server-side procedures that instantiate the framework for the gNMI protocol.

The existing encoding procedure for JSON and JSON_IETF formats push the input data that causes the output values to have premature wraps and other issues, causing loss of data precision.

PROTO encoding allows the querying of applications to retrieve data in scalar (TypedValue) values; each leaf is sent on its own update as per the gNMI specification. This allows customers to have access to float and double values for more accuracy.

PROTO encoding is supported for all the paths that are supported by gNMI. PROTO encoding supports subscribe RPC, but not GET and SET RPCs.

PROTO encoding is part of the gNMI protocol, and is enabled when the gNMI feature is enabled.

The following is a sample subscribeRequest RPC with PROTO:

```
subscribe:
           prefix:
               origin: "legacy"
               elem:
                   name: "mdt-oper-v2:mdt-oper-v2-data"
               >
           >
           subscription:
             path:
              elem:
                  name: "mdt-subscriptions"
                  >
              >
              mode: SAMPLE
              sample_interval: 10000000000
          >
          mode: STREAM
          encoding: PROTO
      >
```

The following is a sample subscribeRequest response with PROTO:

```
update: <
  timestamp: 1652408966709576000
  prefix: <
    origin: "openconfig"
    elem: <
      name: "oc-if:interfaces"
    >
    elem: <
      name: "interface"
      key: <
        key: "name"
        value: "*"
      >
    >
    elem: <
      name: "state"
    >
  update: <
    path: <
      origin: "openconfig"
```

```
      elem: <
        name: "interfaces"
      >
      elem: <
        name: "interface"
        key: <
          key: "name"
          value: "GigabitEthernet3"
        >
      >
      elem: <
        name: "ethernet"
      >
      elem: <
        name: "config"
      >
    >
    val: <
      string_val: "{
\"mac-address\":\"00:0c:29:29:42:e9\”
  }"
    >
  >
update: <
    path: <
      origin: "openconfig"
      elem: <
        name: "interfaces"
      >
      elem: <
        name: "interface"
        key: <
          key: "name"
          value: "GigabitEthernet3"
        >
      >
      elem: <
        name: "ethernet"
      >
      elem: <
        name: "config"
      >
    >
    val: <
      bool_val: "{
\",\"auto-negotiate\":true
  }"
    >
  >
update: <
    path: <
      origin: "openconfig"
      elem: <
        name: "interfaces"
      >
      elem: <
        name: "interface"
        key: <
          key: "name"
          value: "GigabitEthernet3"
        >
      >
      elem: <
        name: "ethernet"
```

```
      >
      elem: <
        name: "config"
      >
    >
    val: <
      bool_val: "{
\"enable-flow-control\":true
  }"
    >
  >
```

Errors are reported using the *status.proto* message in the RPC return message.

gNMI GET Request

The gNMI Get RPC specifies how to retrieve one or more of the configuration attributes, state attributes, derived state attributes, or all attributes associated with a supported mode from a date tree. A GetRequest is sent from a client to the target to retrieve values from the data tree. A GetResponse is sent in response to a GetRequest.

GetRequest JSON Structure

The following is a sample GetRequest JSON structure. Both the GetRequest and GetResponse are displayed.

GetRequest

```
The following is a path for the
openconfig-interfaces model
++++++++ Sending get request: ++++++++
path {
  elem {
    name: "interfaces"
  }
  elem {
    name: "interface"
    key {
      key: "name"
      value: "Loopback111"
    }
  }
}
```

GetResponse

```
encoding: JSON_IETF
++++++++ Received get response: ++++++++
notification {
  timestamp: 1521699434792345469
  update {
    path {
      elem {
        name: "interfaces"
      }
      elem {
        name: "interface"
        key {
          key: "name"
          value: "\"Loopback111\""
        }
```

```
  }
}


val {
  json_ietf_val: "{\n\t\"openconfig-interfaces:name\":\t\
  "Loopback111\",\n\t\
   "openconfig-interfaces:config\":\t{\n\t\t\
   "openconfig-interfaces:type\":\t\"ianaift:
    softwareLoopback\",\n\t\t\
   "openconfig-interfaces:name\":\t\"Loopback111\",\n\t\t\
   "openconfig-interfaces:enabled\":\t\"true\"\n\t},\n\t\
   "openconfig-interfaces:state\":\t{\n\t\t\
   "openconfig-interfaces:type\":\t\"ianaift:
    softwareLoopback\",\n\t\t\
   "openconfig-interfaces:name\":\t\"Loopback111\",\n\t\t\
   "openconfig-interfaces:enabled\":\t\"true\",\n\t\t\
   "openconfig-interfaces:ifindex\":\t52,\n\t\t\


   "openconfig-interfaces:admin-status\":\t\"UP\",\n\t\t\
   "openconfig-interfaces:oper-status\":\t\"UP\",\n\t\t\
   "openconfig-interfaces:last-change\":\t2018,\n\t\t\
   "openconfig-interfaces:counters\":\t{\n\t\t\t\
   "openconfig-interfaces:in-octets\":\t0,\n\t\t\t\
   "openconfig-interfaces:in-unicast-pkts\":\t0,\n\t\t\t\
   "openconfig-interfaces:in-broadcast-pkts\":\t0,\n\t\t\t\
   "openconfig-interfaces:in-multicast-pkts\":\t0,\n\t\t\t\
   "openconfig-interfaces:in-discards\":\t0,\n\t\t\t\
   "openconfig-interfaces:in-errors\":\t0,\n\t\t\t\
   "openconfig-interfaces:in-unknown-protos\":\t0,\n\t\t\t\
   "openconfig-interfaces:out-octets\":\t0,\n\t\t\t\
   "openconfig-interfaces:out-unicast-pkts\":\t0,\n\t\t\t\
   "openconfig-interfaces:out-broadcast-pkts\":\t0,\n\t\t\t\
   "openconfig-interfaces:out-multicast-pkts\":\t0,\n\t\t\t\
   "openconfig-interfaces:out-discards\":\t0,\n\t\t\t\
   "openconfig-interfaces:out-errors\":\t0,\n\t\t\t\
   "openconfig-interfaces:last-clear\":\t2018\n\t\t},\n\t\t\

   "openconfig-platform:hardware-port\":\t\
   "Loopback111\"\n\t},\n\t\
   "openconfig-interfaces:subinterfaces\":\t{\n\t\t\
   "openconfig-interfaces:index\":\t0,\n\t\t\
   "openconfig-interfaces:config\":\t{\n\t\t\t\
   "openconfig-interfaces:index\":\t0,\n\t\t\t\
   "openconfig-interfaces:name\":\t\"Loopback111\",\n\t\t\t\
   "openconfig-interfaces:enabled\":\t\"true\"\n\t\t},\n\t\t\
   "openconfig-interfaces:state\":\t{\n\t\t\t\
   "openconfig-interfaces:index\":\t0,\n\t\t\t\
   "openconfig-interfaces:name\":\t\"Loopback111.0\",\n\t\t\t\
   "openconfig-interfaces:enabled\":\t\"true\",\n\t\t\t\
   "openconfig-interfaces:admin-status\":\t\"UP\",\n\t\t\t\
   "openconfig-interfaces:oper-status\":\t\"UP\",\n\t\t\t\
   "openconfig-interfaces:last-change\":\t2018,\n\t\t\t\
   "openconfig-interfaces:counters\":\t{\n\t\t\t\t\
   "openconfig-interfaces:in-octets\":\t0,\n\t\t\t\t\
   "openconfig-interfaces:in-unicast-pkts\":\t0,\n\t\t\t\t\
   "openconfig-interfaces:in-broadcast-pkts\":\t0,\n\t\t\t\t\
   "openconfig-interfaces:in-multicast-pkts\":\t0,\n\t\t\t\t\
   "openconfig-interfaces:in-discards\":\t0,\n\t\t\t\t\
   "openconfig-interfaces:in-errors\":\t0,\n\t\t\t\t\

   "openconfig-interfaces:out-octets\":\t0,\n\t\t\t\t\
   "openconfig-interfaces:out-unicast-pkts\":\t0,\n\t\t\t\t\
```

```
        "openconfig-interfaces:out-broadcast-pkts\":\t0,\n\t\t\t\t\
        "openconfig-interfaces:out-multicast-pkts\":\t0,\n\t\t\t\t\
        "openconfig-interfaces:out-discards\":\t0,\n\t\t\t\t\
        "openconfig-interfaces:out-errors\":\t0,\n\t\t\t\t\
        "openconfig-interfaces:last-clear\":\
         t2018\n\t\t\t}\n\t\t},\n\t\t\
        "openconfig-if-ip:ipv6\":\t{\n\t\t\t\
        "openconfig-if-ip:config\":\t\"false\",\n\t\t\t\
        "openconfig-if-ip:state\":\t\"false\"\n\t\t}\n\t}\n}"
    }
  }
}
```

GetRequest on a Leaf Value

The following is a sample GetRequest on a Leaf value. Both the GetRequest and the corresponding GetResponse are displayed.

GetRequest

```
++++++++ Sending get request: ++++++++
path {
  elem {
    name: "interfaces"
  }
  elem {
    name: "interface"
    key {
      key: "name"
      value: "Loopback111"
    }
  }
  elem {
    name: "state"
  }
  elem {
    name: "oper-status"
  }
}
```

GetResponse

```
encoding: JSON_IETF
++++++++ Received get response: ++++++++
notification {
  timestamp: 1521699326012374332
  update {
    path {
      elem {
        name: "interfaces"
      }
      elem {
        name: "interface"
        key {
          key: "name"
          value: "\"Loopback111\""
        }
      }
      elem {
        name: "state"
```

```
      }
      elem {
        name: "oper-status"
      }
    }
    val {
      json_ietf_val: "\"UP\""
    }
  }
}
```

gNMI SetRequest

The Set RPC specifies how to set one or more configurable attributes associated with a supported model. A SetRequest is sent from a client to a target to update the values in the data tree.

SetRequests also support JSON keys, and must contain a YANG-prefix, in which the namespace of the element differs from parent.

For example, the *routed-vlan* element derived from augmentation in openconfig-vlan.yang must be entered as *oc-vlan:routed-vlan*, because it is different from the namespace of the parent node (The parent node prefix is oc-if.).

The total set of deletes, replace, and updates contained in any one SetRequest is treated as a single transaction. If any subordinate element of the transaction fails; the entire transaction is disallowed and rolled back. A SetResponse is sent back for a SetRequest.

Table 1: Example of a SetRequest JSON Structure

SetRequest	SetResponse

`++++++++ Sending set request: ++++++++`
`update {`
`  path {`
`  elem {`
`  name: "interfaces"`
`  }`
`  elem {`
`  name: "interface"`
`  key {`
`  key: "name"`
`  value: "Loopback111"`
`  }`
`  }`
`  elem {`
`  name: "config"`
`  }`
`  }`
`  val {`
`  json_ietf_val:`
`"{\"openconfig-interfaces:enabled\":\"false\"}"`
`  }`
`}`
	`++++++++ Received set response: ++++++++`
	`response {`
	`  path {`
	`  elem {`
	`  name: "interfaces"`
	`  }`
	`  elem {`
	`  name: "interface"`
	`  key {`
	`  key: "name"`
	`  value: "Loopback111"`
	`  }`
	`  }`
	`  elem {`
	`  name: "config"`
	`  }`
	`  }`
	`  op: UPDATE`
	`}`
	`timestamp: 1521699342123890045`

Table 2: Example of a SetRequest on Leaf Value

SetRequest	SetResponse

++++++++ Sending set request: ++++++++
update {
  path {
  elem {
  name: "interfaces"
  }
  elem {
  name: "interface"
  key {
  key: "name"
  value: "Loopback111"
  }
  }
  elem {
  name: "config"
  }
  elem {
  name: "description"
  }
  }
  val {
  json_ietf_val: "\"UPDATE DESCRIPTION\""
  }
}
	++++++++ Received set response: ++++++++
	response {
	  path {
	  elem {
	  name: "interfaces"
	  }
	  elem {
	  name: "interface"
	  key {
	  key: "name"
	  value: "Loopback111"
	  }
	  }
	  elem {
	  name: "config"
	  }
	  elem {
	  name: "description"
	  }
	  }
	  op: UPDATE
	}
	timestamp: 1521699342123890045

gNMI Namespace

A namespace specifies the path prefixing to be used in the *origin* field of a message.

This section describes the namespaces used in Cisco IOS XE Gibraltar 16.10.1 and later releases:

- RFC 7951-specified namespaces: Path prefixes use the YANG module name as defined in RFC 7951.

  The RFC 7951-specified value prefixing uses the YANG module name.

  Value prefixing is not affected by the selected path prefix namespace. The following example shows an RFC 7951-specified value prefix:

  ```
  val {
     json_ietf_val:"{
       "openconfig-interfaces:config": {
         “openconfig-interfaces:description":
           “DESCRIPTION”
       }
     }"
   }
  ```

  An RFC 7951-specified namespace prefixing also uses the YANG module name. For example, the openconfig path to a loopback interface will be

  ```
  /openconfig-interfaces:interfaces/interface[name=Loopback111]/
  ```

  The following example shows a gNMI path with RFC7951 namespacing:

  ```
  path {
    origin: “rfc7951"
    elem {
  ```

```
    name: "openconfig-interface:interfaces"
  }
  elem {
    name: "interface"
    key {
      key: "name"
      value: "Loopback111"
    }
  }
}
```

- Openconfig: No path prefixes are used. These can only be used with a path to an openconfig model.

  The behavior of the Openconfig namespace prefixing is the same when no origin or namespace is provided. For example, the openconfig path to a loopback interface will be

  ```
  /interfaces/interface[name=Loopback111]/
  ```

  The following example shows a gNMI path with an Openconfig namespacing:

  ```
  path {
    origin: "openconfig"
    elem {
      name: "interfaces"
    }
    elem {
      name: "interface"
      key {
        key: "name"
        value: "Loopback111"
      }
    }
  }
  ```

- Blank: Same as the openconfig prefix. This is the default.

  The following example shows a gNMI path with a blank Openconfig namespacing:

  ```
  path {
  elem {
      name: "interfaces"
    }
    elem {
      name: "interface"
      key {
        key: "name"
        value: "Loopback111"
      }
    }
  }
  ```

This section describes the path prefixing used in releases prior to Cisco IOS XE Gibraltar 16.10.1.

Here, path prefixing uses the YANG module prefix as defined in the YANG module definition. For example, the openconfig path to a loopback interface will be

```
/oc-if:interfaces/interface[name=Loopback111]/
```

The following example shows a gNMI Path with with legacy namespacing:

```
path {
  origin: "legacy"
  elem {
    name: "oc-if:interfaces"
  }
```

```
  elem {
    name: "interface"
    key {
      key: "name"
      value: "Loopback111"
    }
  }
}
```

gNMI Wildcards

The gNMI protocol supports wildcards for Get paths. This is the ability to use a wildcards in a path to match multiple elements. These wildcards indicate all elements in a given subtree in the schema.

An *elem* is an element, and it is a value between / characters in an xPath. An *elem* is also available in a gNMI path. For example, the position of a wildcard relative to *elem* names implies that the wildcard stands for an interface, and is interpreted as all interfaces.

There are two types of wildcards; implicit and explicit, and both are supported. Get paths support all types and combinations of path wildcards.

- Implicit wildcards: These expand a list of elements in an element tree. Implicit wildcard occurs when a key value is not provided for elements of a list.

  The following is a sample path implicit wildcard. This wildcard will return the descriptions of all interfaces on a device:

  ```
  path {
    elem {
      name: "interfaces"
    }
    elem {
      name: "interface"
    }
    elem {
      name: "config"
    }
    elem {
      name: "description"
    }
  }
  ```

- Explicit wildcards: Provides the same functionality by

  - Specifying an asterisk (*) for either the path element name or key name.

    The following sample shows a path asterisk wildcard as the key name. This wildcard returns the description for all interfaces on a device.

    ```
    path {
      elem {
        name: "interfaces"
      }
      elem {
        name: "interface"
        key {
         key: "name"
          value: "*"
        }
      }
      elem {
        name: "config"
    ```

```
  }
  elem {
    name: "description"
  }
}
```

The following sample shows a path asterisk wildcard as the path name. This wildcard will return the description for all elements that are available in the Loopback111 interface.

```
path {
  elem {
    name: "interfaces"
  }
  elem {
    name: "interface"
    key {
      key: "name"
      value: "Loopback111"
    }
  }
   elem {
    name: "*"
  }
  elem {
    name: "description"
  }
}
```

- Specifying an ellipsis (...) or a blank entry as element names. These wildcards can expand to multiple elements in a path.

  The following sample shows a path ellipsis wildcard. This wildcard returns all description fields available under /interfaces.

```
path {
  elem {
    name: "interfaces"
  }
  elem {
    name: "..."
  }
  elem {
    name: "description"
  }
}
```

The following is a sample GetRequest with an implicit wildcard. This GetRequest will return the oper-status of all interfaces on a device.

```
path {
  elem {
    name: "interfaces"
    }
  elem {
    name: "interface"
  }
  elem {
    name: "state"
  }
  elem {
    name: "oper-status"
  }
```

```
},
type: 0,
encoding: 4
```

The following is a sample GetResponse with an implicit wildcard:

```
notification {
  timestamp: 1520627877608777450
  update {
    path {
      elem {
        name: "interfaces"
      }
      elem {
        name: "interface"
        key {
          key: "name"
          value: "\"FortyGigabitEthernet1/1/1\""
        }
      }
      elem {
        name: "state"
}
      elem {
        name: "oper-status"
      }
    }
    val {
      json_ietf_val: "\"LOWER_LAYER_DOWN\""
    }
  },

<snip>
…
</snip>

update {
    path {
      elem {
        name: "interfaces"
      }
      elem {
        name: "interface"
        key {
          key: "name"
          value: "\"Vlan1\""
        }
      }
      elem {
        name: "state"
      }
      elem {
        name: "oper-status"
      }
    }
    val {
      json_ietf_val: "\"DOWN\""
    }
  }
}
```

gNMI Configuration Persistence

The gNMI Configuration Persistence feature ensures that all successful configuration changes made through the gNMI SetRequest RPC persists across device restarts. Prior to this feature, the gNMI configuration was stored in the running configuration of a device. And the changes were saved by issuing the **write memory** command, or the SaveConfig NETCONF RPC.

All changes in the running configuration, even if the data was modified by processes other than gNMI, the data is saved to the startup configuration, when the SetRequest RPC is issued.

This feature is enabled by default and cannot be disabled.

gNMI Username and Password Authentication

User credentials, the username and password provide authorization as metadata in each gNMI RPC. The following is a sample gNMI Capabilities RPC that use the username and password:

```
metadata = [('username','admin'), ('password','lab')]
cap_request = gnmi_pb2.CapabilityRequest()
# pass metadata to the gnmi_pb2_grpc.gNMIStub object
secure_stub.Capabilities(cap_request, metadata=metadata)
```

gNXI VRF Restrictions

In Cisco IOS XE 17.18.2, the gNXI gRPC server that runs in the gnmib process can be configured to be reachable by user-configured VRFs. Prior to this release, the gNXI gRPC server was reachable from data ports and the management interface.



Note Restricting access to the gNXI gRPC server to a VRF configured by an administrator, improves security by not allowing the server to respond to traffic for non-matching VRFs. The management VRF can also be configured as a dedicated VRF, and in this case, the server will not be reachable by traffic from the default VRF or any other VRF.

These VRF restrictions are supported only for the secure gNXI server that is configured using the **gnxi vrf** command. You can configure an encrypted secure-server that supports authentication through username/password, client certificate, or a combination of both. Only one secure server is supported, you cannot configure multiple secure gRPC server instances listening on separate VRFs.

If a non-management VRF is configured, the gRPC server will not respond to traffic from the management VRF.

This feature can be disabled through the CLI or YANG model with a network management protocol such as NETCONF, RESTCONF, or gNMI.

The following sample of Cisco-IOS-XE-gnmi-cfg YANG model displays the gNXI VRF restriction sections.

```
module Cisco-IOS-XE-gnmi-cfg {
   yang-version 1.1;
   namespace "http://cisco.com/ns/yang/Cisco-IOS-XE-gnmi-cfg";
   prefix gnmi-cfg;
.
.
```

```
.
   revision 2025-07-01 {
     description
       "Add leaf 'vrf' to configure the gNxI server's VRF";
     reference
       "1.4.0";
     cisco-semver:module-version "1.4.0";
   }
.
.
.
     }
     leaf vrf {
       type string {
         length "1..32";
         pattern "[\\] A-Za-z0-9!\"#%&()*+,\\-./:;<=>^?@\\\\$_`{|}\\[']+";
       }
       description
         "The Virtual Routing and Forwarding (VRF) instance to use for connections to the
gNxI server.";
     }
   }
```

gNMI Error Messages

When errors occur, gNMI returns descriptive error messages. The following section displays some gNMI error messages.

The following sample error message is displayed when the path is invalid:

```
gNMI Error Response:
  <_Rendezvous of RPC that terminated with (StatusCode.TERMINATED,
   An error occurred while parsing provided xpath: unknown tag:
   "someinvalidxpath" Additional information: badly formatted or nonexistent path)>
```

The following sample error message is displayed for an unimplemented error:

```
gNMI Error Response:
  <_Rendezvous of RPC that terminated with (StatusCode.UNIMPLEMENTED,
    Requested encoding "ASCII" not supported)>
```

The following sample error message is displayed when the data element is empty:

```
gNMI Error Response:
  <_Rendezvous of RPC that terminated with (StatusCode.NOT_FOUND,
   Empty set returned for path "/oc-if:interfaces/noinfohere")>
```

How to Enable the gNMI Protocol

Perform the following steps to enable the gNMI protocol:

1. Create a set of certs for the gNMI client and device signed by a Certificate Authority (CA).

a. Create Certs with OpenSSL on Linux.

b. Install Certs on a device.

c. Configure gNMI on the device.

d. Verify whether gNMI is enabled and running.

2. Connect the gNMI client using client and root certificates configured in previous steps.

Creating Certs with OpenSSL on Linux

Certs and trustpoint are only required for secure gNMI servers.

The following example shows how to create Certs with OpenSSL on a Linux machine:

```
# Setting up a CA
openssl genrsa -out rootCA.key 2048
openssl req -subj /C=/ST=/L=/O=/CN=rootCA -x509 -new -nodes -key rootCA.key -sha256 -out
rootCA.pem

# Setting up device cert and key
openssl genrsa -out device.key 2048
openssl req -subj /C=/ST=/L=/O=/CN=<hostnameFQDN> -new -key device.key -out device.csr
openssl x509 -req -in device.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out
device.crt -sha256
openssl pkcs12 -export -out mycert.pfx -inkey device.key -in device.crt -certfile rootCA.pem
 -aes128
Enter Export Password:cisco
Verifying - Enter Export Password:cisco

# Setting up client cert and key
openssl genrsa -out client.key 2048
openssl req -subj /C=/ST=/L=/O=/CN=gnmi_client -new -key client.key -out client.csr
openssl x509 -req -in client.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out
client.crt -sha256
```

Installing Certs on a Device Through the CLI

The following example show how to install certs on a device:

```
# Send:
Device# configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
Device(config)# crypto pki trustpoint trustpoint1
Device(ca-trustpoint)# revocation-check none
Device(ca-trustpoint)# crypto pki import AESTEST pkcs12 bootflash:mycert.pfx password cisco
% Importing pkcs12...
Source filename [mycert.pfx]?
Reading file from bootflash:mycert.pfx
CRYPTO_PKI: Imported PKCS12 file successfully.
Device(config)#
```

Enabling gNMI in Insecure Mode

Note This task is applicable in Cisco IOS XE Amsterdam 17.3.1 and later releases.

In a Day Zero setup, first enable the device in insecure mode, then disable it, and enable the secure mode. To stop gNxI in insecure mode, use the **no gnxi server** command.

Note gNxI insecure and secure servers can run simultaneously on a device.

Note The **gnxi** commands apply to both gNMI and gRPC Network Operations Interface (gNOI) services. gNxI tools are a collection of tools for Network Management that use the gNMI and gNOI protocols.

SUMMARY STEPS

1. **enable**
2. **configure terminal**
3. **gnxi**
4. **gnxi server**
5. **gnxi port** *port-number*
6. **end**
7. **show gnxi state**

DETAILED STEPS

Procedure

	Command or Action	Purpose
Step 1	**enable** **Example:** `Device> enable`	Enables privileged EXEC mode. • Enter your password if prompted.
Step 2	**configure terminal** **Example:** `Device# configure terminal`	Enters global configuration mode.
Step 3	**gnxi** **Example:** `Device(config)# gnxi`	Starts the gNxI process.

	Command or Action	Purpose
Step 4	**gnxi server** **Example:** `Device(config)# gnxi server`	Enables the gNxI server in insecure mode.
Step 5	**gnxi port** *port-number* **Example:** `(Optional) Device(config)# gnxi port 50000`	Sets the gNxI port to listen to. • The default insecure gNxI port is 50052.
Step 6	**end** **Example:** `Device(config)# end`	Exits global configuration mode and returns to privileged EXEC mode.
Step 7	**show gnxi state** **Example:** `Device# show gnxi state`	Displays the status of gNxI interfaces.

Enabling gNMI in Secure Mode

Note This task is applicable in Cisco IOS XE Amsterdam 17.3.1 and later releases.

To stop gNxI in secure mode, use the **no gnxi secure-server** command.

Note gNxI insecure and secure servers can simultaneously run on a device.

Note The **gnxi** commands apply to both gNMI and gRPC Network Operations Interface (gNOI) services. gNxI tools are a collection of tools for Network Management that use the gNMI and gNOI protocols.

SUMMARY STEPS

1. **enable**
2. **configure terminal**
3. **gnxi**
4. **gnxi secure-trustpoint** *trustpoint-name*
5. **gnxi secure-server**
6. **gnxi secure-client-auth**
7. **gnxi secure-port**
8. **end**
9. **show gnxi state**

DETAILED STEPS

Procedure

	Command or Action	Purpose
Step 1	**enable** **Example:** `Device> enable`	Enables privileged EXEC mode. • Enter your password if prompted.
Step 2	**configure terminal** **Example:** `Device# configure terminal`	Enters global configuration mode.
Step 3	**gnxi** **Example:** `Device(config)# gnxi`	Starts the gNxI process.
Step 4	**gnxi secure-trustpoint** *trustpoint-name* **Example:** `Device(config)# gnxi secure-trustpoint trustpoint1`	Specifies the trustpoint and cert set that gNxI uses for authentication.
Step 5	**gnxi secure-server** **Example:** `Device(config)# gnxi secure-server`	Enables the gNxI server in secure mode.
Step 6	**gnxi secure-client-auth** **Example:** `Device(config)# gnxi secure-client-auth`	(Optional) The gNxI process authenticates the client certificate against the root certificate.
Step 7	**gnxi secure-port** **Example:** `Device(config)# gnxi secure-port`	(Optional) Sets the gNxI port to listen to. • The default secure gNxI port is 9339.
Step 8	**end** **Example:** `Device(config)# end`	Exits global configuration mode and returns to privileged EXEC mode.
Step 9	**show gnxi state** **Example:** `Device# show gnxi state`	Displays the status of gNxI servers.

Example

The following is sample output from the **show gnxi state** command:

```
Device# show gnxi state

State          Status
--------------------------------
Enabled        Up
```

Connecting the gNMI Client

The gNMI client is connected by using the client and root certificates that are previously configured.

The following example shows how to connect the gNMI client using Python:

```
# gRPC Must be compiled in local dir under path below:
>>> import sys
>>> sys.path.insert(0, "reference/rpc/gnmi/")
>>> import grpc
>>> import gnmi_pb2
>>> import gnmi_pb2_grpc
>>> gnmi_dir = '/path/to/where/openssl/creds/were/generated/'

# Certs must be read in as bytes
>>> with open(gnmi_dir + 'rootCA.pem', 'rb') as f:
>>>     ca_cert = f.read()
>>> with open(gnmi_dir + 'client.crt', 'rb') as f:
>>>     client_cert = f.read()
>>> with open(gnmi_dir + 'client.key', 'rb') as f:
>>>     client_key = f.read()

# Create credentials object
>>> credentials = grpc.ssl_channel_credentials(root_certificates=ca_cert,
private_key=client_key, certificate_chain=client_cert)

# Create a secure channel:
# Default port is 9339, can be changed on ios device with 'gnxi secure-port ####'
>>> port = 9339
>>> host = <HOSTNAME FQDN>
>>> secure_channel = grpc.secure_channel("%s:%d" % (host, port), credentials)

# Create secure stub:
>>> secure_stub = gnmi_pb2_grpc.gNMIStub(secure_channel)

# Done! Let's test to make sure it works:
>>> secure_stub.Capabilities(gnmi_pb2.CapabilityRequest())
supported_models {
<snip>
}
supported_encodings: <snip>
gNMI_version: "0.4.0"
```

Configuration Examples for the gNMI Protocol

Example: Enabling gNMI in Insecure Mode

Note This example is applicable in Cisco IOS XE Amsterdam 17.3.1 and later releases.

The following example shows how to enable the gNxI server in insecure mode:

```
Device> enable
Device# configure terminal
Device(config)# gnxi
Device(config)# gnxi server
Device(config)# gnxi port 50000 <The default port is 50052.>
Device(config)# end
Device#
```

Example: Enabling gNMI in Secure Mode

Note This example is applicable in Cisco IOS XE Amsterdam 17.3.1 and later releases.

The following example shows how to enable the gNxI server in secure mode:

```
Device> enable
Device# configure terminal
Device(config)# gnxi
Device(config)# gnxi secure-trustpoint trustpoint1
Device(config)# gnxi secure-server
Device(config)# gnxi secure-client-auth
Device(config)# gnxi secure-port 50001 <The default port is 9339.>
Device(config)# end
Device#
```

Additional References for the gNMI Protocol

Related Documents

Related Topic	Document Title
DevNet	https://developer.cisco.com/site/ios-xe/
gNMI	https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-specification.md

Related Topic	Document Title
gNMI path encoding	https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-path-conventions.md

Standards and RFCs

Standard/RFC	Title
RFC 7951	JSON Encoding of Data Modeled with YANG

Technical Assistance

Description	Link
The Cisco Support website provides extensive online resources, including documentation and tools for troubleshooting and resolving technical issues with Cisco products and technologies. To receive security and technical information about your products, you can subscribe to various services, such as the Product Alert Tool (accessed from Field Notices), the Cisco Technical Services Newsletter, and Really Simple Syndication (RSS) Feeds. Access to most tools on the Cisco Support website requires a Cisco.com user ID and password.	http://www.cisco.com/support

Feature Information for the gNMI Protocol

The following table provides release information about the feature or features described in this module. This table lists only the software release that introduced support for a given feature in a given software release train. Unless noted otherwise, subsequent releases of that software release train also support that feature.

Use Cisco Feature Navigator to find information about platform support and Cisco software image support. To access Cisco Feature Navigator, go to www.cisco.com/go/cfn. An account on Cisco.com is not required.

Table 3: Feature Information for the gNMI Protocol

Feature Name	Release	Feature Information
gNMI Protocol	Cisco IOS XE Fuji 16.8.1a	This feature describes the model-driven configuration and retrieval of operational data using the gNMI capabilities, GET and SET RPCs. This feature was implemented on the following platforms: • Cisco Catalyst 9300 Series Switches • Cisco Catalyst 9400 Series Switches • Cisco Catalyst 9500 Series Switches
	Cisco IOS XE Gibraltar 16.10.1	In Cisco IOS XE Gibraltar 16.10.1, this feature was implemented on Cisco Catalyst 9500-High Performance Series Switches.
	Cisco IOS XE Gibraltar 16.11.1	In Cisco IOS XE Gibraltar 16.11.1, this feature was implemented on Cisco Catalyst 9600 Series Switches.
	Cisco IOS XE Gibraltar 16.12.1	In Cisco IOS XE Gibraltar 16.12.1, this feature was implemented on the following platforms: • Cisco Catalyst 9200 and 9200L Series Switches • Cisco Catalyst 9300L SKUs • Cisco cBR-8 Converged Broadband Router
	Cisco IOS XE Amsterdam 17.1.1	In Cisco IOS XE Amsterdam 17.1.1, this feature was implemented on the following platforms: • Cisco ASR 900 Series Aggregation Services Routers • Cisco ASR 920 Series Aggregation Services Router • Cisco Network Convergence System 520 Series • Cisco Network Convergence System 4200 Series
	Cisco IOS XE Amsterdam 17.2.1r	In Cisco IOS XE Amsterdam 17.2.1r, this feature was implemented on Cisco ASR 1000 Series Aggregation Services Routers.

Feature Name	Release	Feature Information
	Cisco IOS XE Cupertino 17.8.1	In Cisco IOS XE Cupertino 17.8.1, this feature was implemented on the following platforms: • Cisco Catalyst 9800-CL Wireless Controllers • Cisco Catalyst 9800-40 Wireless Controllers • Cisco Catalyst 9800-80 Wireless Controllers
gNMI IPv6 Support	Cisco IOS XE Dublin 17.10.1	gNMI IPv6 support was enabled in Cisco IOS XE Dublin 17.10.1. This feature was implemented on the following platforms: • Cisco Catalyst 9200 and 9200L Series Switches • Cisco Catalyst 9300, 9300L, and 9300X Series Switches • Cisco Catalyst 9400 Series Switches • Cisco Catalyst 9500 and 9500 High-Performance Series Switches • Cisco Catalyst 9600 Series Switches
gNMI Username and Password Authentication	Cisco IOS XE Gibraltar 16.12.1	The Username and Password Authentication feature was added to the gNMI protocol. This feature is supported on all IOS XE platforms that support gNMI.
gNMI Configuration Persistence	Cisco IOS XE Amsterdam 17.3.1	All successful configuration changes made through the gNMI SetRequest RPC persists across device restarts. This feature was implemented on the following platforms: • Cisco Catalyst 9200 Series Switches • Cisco Catalyst 9300 Series Switches • Cisco Catalyst 9400 Series Switches • Cisco Catalyst 9500 Series Switches • Cisco Catalyst 9600 Series Switches

Feature Name	Release	Feature Information
gNOI Certificate Management	Cisco IOS XE Amsterdam 17.3.1	The gNOI Certificate Management Service provides RPCs to install, rotate, get certificate, revoke certificate, and generate certificate signing request. This feature was implemented on the following platforms: • Cisco Catalyst 9200 Series Switches • Cisco Catalyst 9300 Series Switches • Cisco Catalyst 9400 Series Switches • Cisco Catalyst 9500 Series Switches • Cisco Catalyst 9600 Series Switches
gNXI VRF Restriction	Cisco IOS XE 17.18.2	The gNXI gRPC server that runs in the gnmib process can be configured to be reachable by user-configured VRFs. Prior to this release, the gNXI gRPC server was reachable from data ports and the management interface. These VRF restrictions are supported only for secure gNXI servers that are configured using the **gnxi secure-vrf** command. This feature was implemented on the following platforms: • Cisco Catalyst 9200 Series Switches • Cisco Catalyst 9300 Series Switches • Cisco Catalyst 9400 Series Switches • Cisco Catalyst 9500 Series Switches • Cisco Catalyst 9600 Series Switches

Feature Name	Release	Feature Information
Named Method List	Cisco IOS XE Cupertino 17.9.1	

Feature Name	Release	Feature Information
		With the introduction of the Named Method List feature, it is possible to use a custom method-list name for authentication and authorization, without changing the existing AAA configuration of a device. Prior to this feature, only the default method-list was supported. For more information, see the NETCONF Protocol chapter. This feature was implemented on the following platforms: • Cisco 1000 Series Integrated Services Routers • Cisco 4000 Series Integrated Services Routers • Cisco ASR 900 Series Aggregation Services Routers • Cisco ASR 920 Series Aggregation Services Routers • Cisco ASR 1000 Aggregation Services Routers • Cisco Catalyst 8200 Series Edge Platforms • Cisco Catalyst 8300 Series Edge Platforms • Cisco Catalyst 8500 Series and 8500L Series Edge Platforms • Cisco Catalyst 9200 Series Switches • Cisco Catalyst 9300 Series Switches • Cisco Catalyst 9400 Series Switches • Cisco Catalyst 9500 Series Switches • Cisco Catalyst 9600 Series Switches • Cisco Catalyst 9800 Series Wireless Controllers • Cisco Cloud Services Router 1000V Series • Cisco Network Convergence System 520 Series • Cisco Network Convergence System

Feature Name	Release	Feature Information
		4200 Series
PROTO Encoding	Cisco IOS XE Dublin 17.11.1	gNMI protocol supports PROTO encoding. The *gnmi.proto* file represents the blueprint for generating a complete set of client and server-side procedures that represents the framework for the gNMI protocol. This feature was implemented on the following platforms: • Cisco Catalyst 9200, 9200L, and 9200X Series Switches • Cisco Catalyst 9300, 9300L, and 9300X Series Switches • Cisco Catalyst 9400 and 9400X Series Switches • Cisco Catalyst 9500, 9500 High-Performance, and 9500X Series Switches • Cisco Catalyst 9600 Series Switches