```
Accept-Encoding: gzip, deflate
Connection: keep-alive

HTTP/1.1 200 OK
Server: nginx
Date: Fri, 13 Dec 2019 21:59:25 GMT
Content-Type: application/yang-data+json
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Pragma: no-cache

{
  "ietf-interfaces:interface": [
    {
      "name": "GigabitEthernet1",
      "description": "MANAGEMENT INTERFACE - DON'T TOUCH ME",
      "type": "iana-if-type:ethernetCsmacd",
      "enabled": true,
      "ietf-ip:ipv4": {
        "address": [
          {
            "ip": "10.10.20.48",
            "netmask": "255.255.255.0"
          }
        ]
      },
      "ietf-ip:ipv6": {}
    },
    {
      "name": "GigabitEthernet2",
      "description": "Network Interface",
      "type": "iana-if-type:ethernetCsmacd",
      "enabled": true,
      "ietf-ip:ipv4": {},
      "ietf-ip:ipv6": {}
    },
    {
      "name": "GigabitEthernet3",
      "description": "Network Interface",
      "type": "iana-if-type:ethernetCsmacd",
      "enabled": false,
      "ietf-ip:ipv4": {},
      "ietf-ip:ipv6": {}
    }
  ]
}
```

Finally, a client can use the HEAD method in place of the GET method in order to retrieve the resource metadata only. The response message for a HEAD request is exactly the same response if a GET method is used, minus the message body.

Editing Data: POST, PUT, PATCH, and DELETE

A client can use the POST method to either create a data resource or invoke an operations resource. The resource to be created is a child resource to the target resource identified by the request URI. The resource representation goes into the message body. In Example 14-57, interface Loopback123 is created under the container interfaces as an instance of the list interface.

Example 14-57 *Creating Interface Loopback123 by Using the POST Method*

```
! Interface list from the CLI before sending the POST request
csr1000v-1#show ip interface brief
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet1       10.10.20.48     YES NVRAM  up                    up
GigabitEthernet2       unassigned      YES DHCP   up                    up
GigabitEthernet3       unassigned      YES NVRAM  administratively down down

! POST request to create interface Loopback123
POST /restconf/data/ietf-interfaces:interfaces/ HTTP/1.1
Content-Type: application/yang-data+xml
Authorization: Basic ZGV2ZWxvcGVyOkMxc2NvMTIzNDU=
User-Agent: PostmanRuntime/7.20.1
Accept: */*
Cache-Control: no-cache
Host: ios-xe-mgmt-latest.cisco.com:9443
Accept-Encoding: gzip, deflate
Content-Length: 560
Connection: keep-alive

<interface xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"
xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <name>Loopback123</name>
    <description>Creating a Loopback interface using RESTCONF and POST</description>
    <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-
type">ianaift:softwareLoopback</type>
    <enabled>true</enabled>
    <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
        <address>
            <ip>10.123.123.123</ip>
            <netmask>255.255.255.255</netmask>
        </address>
    </ipv4>
</interface>

HTTP/1.1 201 Created
Server: nginx
Date: Fri, 13 Dec 2019 23:20:08 GMT
Content-Type: text/html
Content-Length: 0
Location: https://ios-xe-mgmt-latest.cisco.com/restconf/data/ietf-
interfaces:interfaces/interface=Loopback123
Connection: keep-alive
Last-Modified: Fri, 13 Dec 2019 23:20:08 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Etag: 1576-279208-528007
Pragma: no-cache

! Interface list from the CLI after sending the POST request
csr1000v-1#show ip interface brief
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet1       10.10.20.48     YES NVRAM  up                    up
GigabitEthernet2       unassigned      YES DHCP   up                    up
GigabitEthernet3       unassigned      YES NVRAM  administratively down down
Loopback123            10.123.123.123  YES other  up                    up
```

If the resource is created successfully, as in Example 14-57, the response has a "201 Created" status line with a Location header field pointing to the newly created resource, as highlighted in the example.

The POST method creates a new resource, and if that resource already exists, a "409 Conflict" status line is returned. Re-sending the request from Example 14-57 has this exact effect, as shown in Example 14-58.

Example 14-58 *Response Received When Attempting to Create Interface Loopback123 When It Already Exists*

```
HTTP/1.1 409 Conflict

Server: nginx

Date: Thu, 26 Dec 2019 16:38:32 GMT

Content-Type: application/yang-data+xml

Transfer-Encoding: chunked

Connection: keep-alive

Cache-Control: private, no-cache, must-revalidate, proxy-revalidate

Vary: Accept-Encoding

Pragma: no-cache


<errors xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf">

  <error>

    <error-message>object already exists:

/if:interfaces/if:interface[if:name='Loopback123']</error-message>

    <error-path>/ietf-interfaces:interfaces</error-path>

    <error-tag>data-exists</error-tag>

    <error-type>application</error-type>

  </error>

</errors>
```

If the request body contains more than one instance of a list or leaf list, a "400 Bad Request" response is returned. Accordingly, this will be the response received if you attempt to create more than one interface in the same request message.

The other use of the POST method is to invoke an operations resource, as described earlier in this chapter, in the section "The Operations Resource." In that case, the message body of the request message can be used to include the input parameters of the operation (**rpc** or **action**), if any. Similarly, the message body of the response message is used to communicate the output parameters of the operation, if any.

The PUT method also creates a resource, but unlike the POST method, PUT *replaces* a resource if it finds that it already exists. When a PUT method creates a resource, a "201 Created" response is sent back to the client. If the resource already exists and is replaced by the PUT request, a "204 No Content" response is sent back to the client.

Another subtle difference between the POST and PUT methods is that the URI in a POST request is that of the parent resource under which the new resource is created, whereas the target URI in a PUT request is that of the newly created resource itself.

Say that you want to create interface Loopack124 by using PUT. The URI to use is not https://ios-xe-mgmt-latest.cisco.com/restconf/data/ietf-interfaces:interfaces/, as it would be with the POST method. If you use that URI, you get a "400 Bad Request" response. The correct URI to use is https://ios-xe-mgmt-latest.cisco.com/restconf/data/ietf-interfaces:interfaces/interface=Loopback124, as shown in Example 14-59.

Example 14-59 *Creating Interface Loopback124 by Using the PUT Method*

```
! Interface list before the PUT request is sent

csr1000v-1#show ip interface brief

Interface              IP-Address      OK? Method Status                Protocol

GigabitEthernet1       10.10.20.48     YES NVRAM  up                    up

GigabitEthernet2       unassigned      YES DHCP   up                    up

GigabitEthernet3       unassigned      YES NVRAM  administratively down down

Loopback123            10.123.123.123  YES other  up                    up


! PUT request to create interface Loopback124

PUT /restconf/data/ietf-interfaces:interfaces/interface=Loopback124 HTTP/1.1

Content-Type: application/yang-data+xml

Authorization: Basic ZGV2ZWxvcGVyOkMxc2NvMTIzNDU=

User-Agent: PostmanRuntime/7.20.1

Accept: */*

Cache-Control: no-cache

Host: ios-xe-mgmt-latest.cisco.com:9443

Accept-Encoding: gzip, deflate

Content-Length: 560
```

```
Connection: keep-alive


<interface xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"

xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">

    <name>Loopback124</name>

    <description>Creating a Loopback interface using RESTCONF and POST</description>

    <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-

type">ianaift:softwareLoopback</type>

    <enabled>true</enabled>

    <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">

        <address>

            <ip>10.123.123.124</ip>

            <netmask>255.255.255.255</netmask>

        </address>

    </ipv4>

</interface>


HTTP/1.1 201 Created

Server: nginx

Date: Fri, 13 Dec 2019 23:49:19 GMT

Content-Type: text/html

Content-Length: 0

Location: https://ios-xe-mgmt-latest.cisco.com/restconf/data/ietf-

interfaces:interfaces/interface=Loopback124

Connection: keep-alive

Last-Modified: Fri, 13 Dec 2019 23:49:18 GMT

Cache-Control: private, no-cache, must-revalidate, proxy-revalidate

Etag: 1576-280958-904019

Pragma: no-cache


! Interface list after the PUT request is sent

csr1000v-1#show ip interface brief

Interface              IP-Address      OK? Method Status                Protocol

GigabitEthernet1       10.10.20.48     YES NVRAM  up                    up

GigabitEthernet2       unassigned      YES DHCP   up                    up

GigabitEthernet3       unassigned      YES NVRAM  administratively down down

Loopback123            10.123.123.123  YES other  up                    up

Loopback124            10.123.123.124  YES other  up                    up
```

Note that the PUT method is not intended to be used to edit a resource. The PUT method replaces a resource with the data in the message body. The PUT method removes the old configuration and places the new configuration in the datastore, effectively losing all the old configuration.

The primary way to edit a resource is using the PATCH method. The PATCH method enables you to selectively edit the target resource. Unlike the PUT and POST methods, which either create or completely replace a resource, the PATCH method operates by merging the contents in the HTTP request message body with the target resource.

The PATCH method has several flavors. The one covered in this section is named the *plain PATCH*.

Say, for example, that you need to edit the description on an interface but don't want to replace the whole interface configuration—perhaps because you do not need to worry about what the current configuration is, or you don't know what this configuration is—and you need an extra operation to retrieve this data. In this case, the PATCH method would be a better fit than the PUT or POST methods.

Example 14-60 shows the description on interface GigabitEthernet3 before and after using the PATCH method to change the interface description.

Example 14-60 *Using the PATCH Method to Edit the Interface Description*

```
! Interface description BEFORE applying the PATCH method

csr1000v-1#show run interface Gig3

Building configuration...


Current configuration : 126 bytes
```

```
!
interface GigabitEthernet3
 description Interface Description BEFORE the PATCH method
 no ip address
 negotiation auto
end

! HTTP request message using the PATCH method and the resulting response
PATCH /restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet3 HTTP/1.1
Content-Type: application/yang-data+xml
Authorization: Basic ZGV2ZWxvcGVyOkMxc2NvMTIzNDU=
User-Agent: PostmanRuntime/7.19.0
Accept: */*
Cache-Control: no-cache
Host: ios-xe-mgmt-latest.cisco.com:9443
Accept-Encoding: gzip, deflate
Content-Length: 216
Connection: keep-alive

<interface
    xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"
    xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <name>GigabitEthernet3</name>
    <description>Interface Description AFTER the PATCH method</description>
</interface>

HTTP/1.1 204 No Content
Server: nginx
Date: Mon, 18 Nov 2019 07:01:55 GMT
Content-Type: text/html
Content-Length: 0
Connection: keep-alive
Last-Modified: Mon, 18 Nov 2019 07:01:55 GMT
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Etag: 1574-60515-423386
Pragma: no-cache

! Interface description AFTER applying the PATCH method
csr1000v-1#show run interface Gig3
Building configuration...

Current configuration : 125 bytes
!
interface GigabitEthernet3
 description Interface Description AFTER the PATCH method
 no ip address
 negotiation auto
end
```

If the PATCH method is used to edit a child resource (description) of the target resource (interface GigabitEthernet3), as in Example 14-60, keep in mind the following points:

• A plain PATCH can be used to create or update but not delete a child resource.

• If the target resource is a YANG leaf list, the PATCH method must not change the value of the leaf list instance.

• If the target resource is a YANG list instance, the key leaf values in the message body must match the key leaf values in the URI. In addition, the PATCH method must not be used to change the key leaf values.

If we apply the first restriction to Example 14-60, we see that the plain PATCH method cannot be used to delete the interface description; it can only update it. It can also be used to create an interface description if this description does not already exist.

Now consider the third restriction in the list and note that Interface GigabitEthernet3 in Example 14-60 is an instance of the list interface, whose key is the leaf name, as defined in the YANG module ietf-interfaces. Although you can use the PATCH request without including the interface name in the body, if you do include it, the value of the element <name> in the message body must be equal to the name of the interface in the URI, as you can see in Example 14-60. In addition, you cannot change the value of the key, so you cannot change the name of the interface in a PATCH request message. This might seem obvious for GigabitEthernet interfaces whose names cannot be changed anyway, but it is not so obvious for other types, such as loopback, tunnel, or VLAN interfaces.

An important point to keep in mind is that when using the PATCH method to edit a resource, the Content-Type header in the request message must match the data encoding in the message body (in this case, application/yang-data+xml). If a response has an incorrect or missing value in the Content-Type header field, the response will indicate an error, as shown in Example 14-61.

Example 14-61 *Error in Response Message Due to the Use of the Wrong Media Type in the Content-Type Header*

```
HTTP/1.1 415 Unsupported Media Type

Server: nginx

Date: Mon, 18 Nov 2019 07:51:46 GMT

Content-Type: application/yang-data+xml

Transfer-Encoding: chunked

Connection: keep-alive

Cache-Control: private, no-cache, must-revalidate, proxy-revalidate

Vary: Accept-Encoding

Pragma: no-cache


<errors xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf">

  <error>

    <error-message>Unsupported media type: text/plain ; Should be one of:

application/yang-data+xml, application/yang-data+json, application/yang-patch+xml,

application/yang-patch+json.</error-message>

    <error-tag>malformed-message</error-tag>

    <error-type>application</error-type>

  </error>

</errors>
```

A successful PATCH operation results in a "200 OK" or "204 No Content" message, depending on whether the response message includes a body.

Finally, the DELETE method is used to delete the target resource. To delete interface Loopback124 on the IOS XE sandbox, you send to the device the request in Example 14-62.

Example 14-62 *Using the DELETE Method to Delete Interface Loopback124*

```
DELETE /restconf/data/ietf-interfaces:interfaces/interface=Loopback124 HTTP/1.1

Content-Type: application/yang-data+xml

Authorization: Basic ZGV2ZWxvcGVyOkMxc2NvMTIzNDU=

User-Agent: PostmanRuntime/7.20.1

Accept: */*

Cache-Control: no-cache

Host: ios-xe-mgmt-latest.cisco.com:9443

Accept-Encoding: gzip, deflate

Content-Length: 0

Connection: keep-alive


HTTP/1.1 204 No Content

Server: nginx

Date: Sat, 14 Dec 2019 00:08:43 GMT

Content-Type: text/html

Content-Length: 0

Connection: keep-alive

Last-Modified: Sat, 14 Dec 2019 00:08:42 GMT

Cache-Control: private, no-cache, must-revalidate, proxy-revalidate

Etag: 1576-282122-423882

Pragma: no-cache
```

A “204 No Content” response indicates that the resource has been successfully deleted.

Query Parameters

Query parameters provide you with more control over what you retrieve when you use the HEAD or GET methods or more control over what effect the POST or PUT methods have on a resource. A URI with query parameters has the following general format:

https://{device_address}**:**{port}**/**{api_resource}**/**{full-path-to-resource}**?**{paramter_1}**=**{value_1}[**&**{parameter_2}**=**{value_2}][**&**..][**&**{parameter_n}**=**{value_n}]

Vendors can define their own parameters, but Section 4.8 of RFC 8040 defines the following nine query parameters:

• **content**: This parameter is used to select whether a GET request retrieves configuration and/or non-configuration data, using one of three values: config, nonconfig, and all. The default value is all.

• **depth**: This parameter is used to select the number of levels down the resource hierarchy retrieved by a GET request; unbounded is the default value.

• **fields**: This parameter is used in GET requests to retrieve only a subset of the target resource child fields/nodes.

• **filter**: This parameter is used in a GET request to a server event stream resource to filter which notifications are received from that server, using XPath 1.0 expressions. Notifications and event streams are covered in Section 6 of RFC 8040.

• **insert**: When a resource is created using the POST or PUT methods, and that resource is part of a list or leaf list that is ordered by the user, this query parameter allows the user to specify where the new resource fits in the list. This parameter takes one of four values: first, last, before, and after. The default value is last.

• **point**: This parameter works with the insert query parameter to specify where to insert a list or leaf list entry when the insert parameter is either before or after. The format of the value assigned to the point parameter is the same as the URI string.

• **start-time**: This parameter takes a value formatted as a date and a time and is used in GET requests to trigger the notification replay feature defined in RFC 5277. The replay should start at the time specified in the parameter value.

• **stop-time**: This parameter works with the start-time parameter and indicates the last (newest) notification that should be received.

• **with-defaults**: When servers support the with-defaults capability, this query parameter used in a GET request tells the server how to respond to the client request when the data retrieved contains nodes that have a default value, as defined in the corresponding YANG module. This parameter takes one of four values: report-all, trim, explicit, and report-all-tagged. The default value depends on the server.

As you can see from the general format of the URI with query parameters, one or more parameters may be used in the same URI, separated using the ampersand (&) symbol, as long as each parameter does not appear more than once. The parameters may appear in any order. Which query parameters are allowed for a resource may depend on the particular resource, but typically, only the resource type is the important factor. Query parameters and their values are case sensitive.

If a parameter is given an invalid value, or if a parameter is used with a resource or method that does not support that parameter or if a parameter appears more than once in a URI, the server returns a “400 Bad Request” response.

The content query parameter tells the server whether to respond to a client GET request with configuration data and/or non-configuration data. The three possible values of this parameter are config, nonconfig, and all. The default value when the parameter is not used is all. Therefore, sending a GET request to the URI https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data?content=config would return only the configuration data in the datastore on the IOS XE sandbox.

The depth query parameter tells the server how many levels down the resource hierarchy it needs to send back to the client in response to a GET request. The value of the depth parameter should be an integer. Different implementations define a depth of 1 differently. Some define depth=1 as the target resource level, and some define that depth level as the first level of child resources. Depth may also be set to unbounded, which is the default value if the depth parameter is omitted altogether. It basically means that the resource and all resources under it, at all levels, will be retrieved.

Example 14-63 shows the message body of the response received when sending a GET request to the URI https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet3?depth=1 and then changing the depth to 2 and finally to unbounded. Notice the different levels of child nodes in each case.

Example 14-63 *Responses received when sending a GET Request with Different Depth Levels*

```
! Response received when sending a GET request using depth=1

<interface xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"

xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">

    <name>GigabitEthernet3</name>

    <description>Network Interface</description>

    <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-

type">ianaift:ethernetCsmacd</type>

    <enabled>false</enabled>

    <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"/>

    <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip"/>

</interface>


! Response received when sending a GET request using depth=2

<interface xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"

xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">

    <name>GigabitEthernet3</name>
```

```
    <description>Network Interface</description>
    <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-
type">ianaift:ethernetCsmacd</type>
    <enabled>false</enabled>
    <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
        <address/>
        <address/>
    </ipv4>
    <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
        <ipv6-router-advertisements xmlns="urn:ietf:params:xml:ns:yang:ietf-ipv6-
unicast-routing"/>
    </ipv6>
</interface>


! Response received when sending a GET request using depth=unbounded ###
<interface xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"
xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <name>GigabitEthernet3</name>
    <description>Network Interface</description>
    <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-
type">ianaift:ethernetCsmacd</type>
    <enabled>true</enabled>
    <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
        <address>
            <ip>10.0.0.3</ip>
            <netmask>255.255.255.0</netmask>
        </address>
        <address>
            <ip>10.0.1.3</ip>
            <netmask>255.255.255.0</netmask>
        </address>
    </ipv4>
    <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
  </ipv6>
</interface>
```

With the fields query parameter, you have the option of selecting which child elements of a resource are retrieved by a GET request. Example 14-64 shows how to retrieve only the child element <enabled> for interface GigabitEthernet3.

Example 14-64 *Using the Fields Query Parameter to Retrieve the <enabled> Child Node Only*

```
! Sending a GET request to retrieve only the <enabled> element
GET /restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet3?fields=enabled
HTTP/1.1
Authorization: Basic ZGV2ZWxvcGVyOkMxc2NvMTIzNDU=
User-Agent: PostmanRuntime/7.21.0
Accept: */*
Cache-Control: no-cache
Host: ios-xe-mgmt-latest.cisco.com:9443
Accept-Encoding: gzip, deflate
Connection: keep-alive


! Resulting response message
HTTP/1.1 200 OK
Server: nginx
Date: Thu, 26 Dec 2019 11:16:22 GMT
Content-Type: application/yang-data+xml
```

```
Transfer-Encoding: chunked

Connection: keep-alive

Cache-Control: private, no-cache, must-revalidate, proxy-revalidate

Pragma: no-cache


<interface xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"

xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">

  <enabled>true</enabled>

</interface>
```

As you can see, other fields do not show up. What if you need to display the interface name as well? You can specify more than one field, separated by semicolons (;), as shown in Example 14-65.

Example 14-65 *Using a Semicolon to Specify More Than One Value for the Fields Query Parameter*

```
! Sending a GET request to retrieve more than one child field

GET /restconf/data/ietf-

interfaces:interfaces/interface=GigabitEthernet3?fields=name;enabled HTTP/1.1

Authorization: Basic ZGV2ZWxvcGVyOkMxc2NvMTIzNDU=

User-Agent: PostmanRuntime/7.21.0

Accept: */*

Cache-Control: no-cache

Host: ios-xe-mgmt-latest.cisco.com:9443

Accept-Encoding: gzip, deflate

Connection: keep-alive


### Resulting response message ###

HTTP/1.1 200 OK

Server: nginx

Date: Thu, 26 Dec 2019 11:19:03 GMT

Content-Type: application/yang-data+xml

Transfer-Encoding: chunked

Connection: keep-alive

Cache-Control: private, no-cache, must-revalidate, proxy-revalidate

Pragma: no-cache


<interface xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"

xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">

  <name>GigabitEthernet3</name>

  <enabled>true</enabled>

</interface>
```

Example 14-66 shows a more complex expression, where the interface name, status, and IP address are retrieved by sending a GET request to the URI https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data/ietf-interfaces:interfaces?fields=interface(name;enabled;ietf-ip:ipv4/address/ip).

Example 14-66 *Using a Complex Expression for the Field Query Parameter to Retrieve More Than One Child Element*

```
! Sending a GET request using a complex expression for the fields query parameter

GET /restconf/data/ietf-interfaces:interfaces?fields=interface%28name;enabled;ietf-

ip:ipv4/address/ip%29 HTTP/1.1

Authorization: Basic ZGV2ZWxvcGVyOkMxc2NvMTIzNDU=

User-Agent: PostmanRuntime/7.21.0

Accept: */*

Cache-Control: no-cache

Host: ios-xe-mgmt-latest.cisco.com:9443

Accept-Encoding: gzip, deflate

Connection: keep-alive


### Resulting response message ###
```

```
HTTP/1.1 200 OK
Server: nginx
Date: Thu, 26 Dec 2019 11:27:15 GMT
Content-Type: application/yang-data+xml
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Pragma: no-cache



<interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"
xmlns:if-"urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
        <name>GigabitEthernet1</name>
        <enabled>true</enabled>
        <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
            <address>
                <ip>10.10.20.48</ip>
            </address>
        </ipv4>
    </interface>
    <interface>
        <name>GigabitEthernet2</name>
        <enabled>true</enabled>
        <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
            <address>
                <ip>10.255.255.1</ip>
            </address>
        </ipv4>
    </interface>
    <interface>
        <name>GigabitEthernet3</name>
        <enabled>true</enabled>
        <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
            <address>
                <ip>10.0.0.3</ip>
            </address>
            <address>
                <ip>10.0.1.3</ip>
            </address>
        </ipv4>
    </interface>
</interfaces>
```

Notice that the following conventions are used when constructing an expression to use with the fields query parameter:

• As mentioned earlier, you can specify more than one field by separating the different child node names with semicolons (;). In Example 14-66, the semicolon is used to extract three different fields.

• You can use a slash (/) to explicitly extract a node down a hierarchy without including other nodes. In Example 14-66, only ip is extracted, using the path expression ipv4/address/ip, in the process excluding the mask.

• When a child node is defined in a different module/namespace, you specify the child node as {*namespace*}:{*child-name*} (for example, ietf-ip:ipv4).

• You use parentheses to specify the subnodes under the primary node, such as to specify the three different fields under the interface field.

All rules that apply to URI queries, as discussed in Chapter 7, apply to the query parameters discussed here. For example, notice that the right and left parentheses are encoded to %28 and %29, respectively, in the Example 14-66.

For a description of the remaining query parameters, see Section 4.8 of RFC 8040.

RESTCONF and Python

Because RESTCONF is a RESTful protocol, the same Python libraries discussed in Chapter 7 are used to generate and send RESTCONF requests: the socket module and the urllib, httplib, and requests packages. Example 14-67 shows a Python program that uses the requests package to send a POST request that creates interface Loopback111.

Example 14-67 *Creating Interface Loopback111 by Using the requests Package in Python*

```
import requests
url = 'https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data/ietf-
interfaces:interfaces/'
headers = {'Content-Type': 'application/yang-data+xml'}
payload = '''
   <interface xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"
xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <name>Loopback111</name>
    <description>Creating a Loopback interface using Python</description>
    <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-
type">ianaift:softwareLoopback</type>
    <enabled>true</enabled>
    <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
        <address>
            <ip>10.111.111.111</ip>
            <netmask>255.255.255.255</netmask>
        </address>
    </ipv4>
   </interface>'''
res_obj = requests.post(url,headers=headers,data=payload,auth=('developer',
'C1sco12345'),verify=False)
print('The request headers:','\n',res_obj.request.headers,'\n')
print('The response message body from the server is:','\n',res_obj.text,'\n')
print('The response status code:','\n',res_obj.status_code,'\n')
print('The response headers:','\n',res_obj.headers,'\n')
```

Example 14-68 shows the output from running the script.

Example 14-68 *The Output from the Four Print Statements in the Script in the Previous Example*

```
[NetDev@Server1 Scripts]$ python RESTCONF_requests.py
/usr/local/lib/python3.7/site-packages/urllib3/connectionpool.py:847:
InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate
verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
  InsecureRequestWarning)
The request headers:
 {'User-Agent': 'python-requests/2.22.0', 'Accept-Encoding': 'gzip, deflate', 'Accept':
'*/*', 'Connection': 'keep-alive', 'Content-Type': 'application/yang-data+xml',
'Content-Length': '556', 'Authorization': 'Basic ZGV2ZWxvcGVyOkMxc2NvMTIzNDU='}


The response message body from the server is:



The response status code:
 201


The response headers:
 {'Server': 'nginx', 'Date': 'Fri, 27 Dec 2019 13:53:04 GMT', 'Content-Type':
'text/html', 'Content-Length': '0', 'Location': 'https://ios-xe-mgmt-
latest.cisco.com/restconf/data/ietf-interfaces:interfaces/interface=Loopback111',
'Connection': 'keep-alive', 'Last-Modified': 'Fri, 27 Dec 2019 13:53:04 GMT', 'Cache-
Control': 'private, no-cache, must-revalidate, proxy-revalidate', 'Etag': '1577-454784-
```

```
616529', 'Pragma': 'no-cache'}


[NetDev@Server1 Scripts]$
```

Notice the Location header, whose value is the URI pointing to the location of the newly created resource—in this case, interface Loopback111.

Example 14-69 shows the output from the CLI with the interface list before and after running the script.

Example 14-69 *The Interface List Before and After Running the Script Showing the Newly Created Interface Loopback111*

```
! Interface list BEFORE running the script

csr1000v-1#sh ip int bri

Interface              IP-Address      OK? Method Status                Protocol

GigabitEthernet1       10.10.20.48     YES NVRAM  up                    up

GigabitEthernet2       10.255.255.1    YES other  up                    up

GigabitEthernet3       10.0.0.3        YES manual up                    up

Loopback123            10.123.123.123  YES other  up                    up

csr1000v-1#


! Interface list AFTER running the script

csr1000v-1#show ip int bri

Interface              IP-Address      OK? Method Status                Protocol

GigabitEthernet1       10.10.20.48     YES NVRAM  up                    up

GigabitEthernet2       10.255.255.1    YES other  up                    up

GigabitEthernet3       10.0.0.3        YES manual up                    up

Loopback111            10.111.111.111  YES other  up                    up

Loopback123            10.123.123.123  YES other  up                    up

csr1000v-1#
```

For a full discussion of the requests package and the other Python libraries for working with HTTP and RESTCONF, refer to Chapter 7.

Summary

This chapter covers the first two protocols in the network programmability stack: NETCONF and RESTCONF.

The following are the main takeaways from this chapter for NETCONF:

• At the transport layer, NETCONF requires a session-based, reliable, and secure protocol, and all implementations, at a minimum, must implement NETCONF over SSH.

• At the messages layer, NETCONF defines three types of messages: hello, rpc, and rpc-reply.

• At the operations layer, NETCONF defines several operations for retrieving data, editing data, locking and unlocking datastores, and handling session management.

• At the content layer, NETCONF uses XML exclusively for encoding the message body and follows the hierarchy defined by either an XML Schema Definition (XSD) or a YANG model.

The following are the main takeaways from this chapter for RESTCONF:

• RESTCONF is a RESTful protocol and therefore is not a stateful or session-based protocol.

• At the transport layer, RESTCONF exclusively uses HTTPS.

• At the messages layer, RESTCONF uses HTTP request and response messages.

• At the operations layer, RESTCONF defines hierarchical resources and uses URIs to target these resources. It uses HTTP methods to act on these resources.

• At the content layer, RESTCONF uses XML or JSON for encoding the message body, and it may only follow the hierarchy defined by a YANG model.

Chapter 15. gRPC, Protobuf, and gNMI

In the previous two chapters, you learned about two network management protocols, NETCONF and RESTCONF. These two protocols form the foundation of network programmability and are the two most ubiquitous protocols in today's enterprise networks—for service providers and data centers alike. However, the development of network technologies, including network automation protocols, is never in a frozen state; as new challenges arise, new solutions need to be developed to meet those challenges. This chapter introduces a new protocol that has been to developed to solve some of these newly emerging challenges: gRPC. This protocol relies on an absolutely new data serialization format named Protocol Buffers (shortly Protobuf). Also, the new transport requires a new message set, new specification, which benefits from the transport at most. This role is taken by gNMI, which stands for gRPC Network Management Interface.

Requirements for Efficient Transport

One of the challenges that originated in the data center world and then later became applicable to enterprise and service provider networks as well involves multiple requirements in different dimensions that may seem, initially, contradictory:

• On one hand, there is ever-growing utilization of interfaces in data centers and service provider networks, where typical traffic rates today are on the order of hundreds of gigabits per second. Hence, there is an ever-growing need to reduce any overhead traffic, including management plane traffic, as much as possible.

• On the other hand, there is a need to collect as much operational data as possible from the network elements, including counters, routing protocols states, and the contents of MPLS FIB and MAC address tables. The need extends to analyzing this data in real time—or as close to real time as possible. In other words, there is a need for this telemetry to be continuously streamed from network devices.

• There is an additional complexity associated with streaming telemetry: Telemetry avoids the unnecessary load on network element resources, as well as unnecessary traffic on the transport media caused by request/response operations, required if streaming telemetry is not used. Originally, the concept of subscriptions was covered in RFC 5277, which addresses NETCONF event notifications. However, early production implementations of NETCONF did not implement subscriptions. Later, Cisco extended the capability of NETCONF subscriptions to some Cisco IOS XE platforms. (Refer to RFC 8640 for further details.) However, subscriptions involve huge administrative overhead on the wire due to XML encapsulation and are therefore are not very efficient.

These requirements collectively drove research for a solution that would both fulfill the industry requirement and mitigate the shortcomings of the solutions implemented then. As you have already learned, network automation and programmability employs similar technologies and protocols to those used for application development and interaction. For example, NETCONF was inspired by SOAP/XML and is based on XML, and RESTCONF was inspired by and based on REST. To network programmability researchers, this indicated that a solution probably existed in the applications domain and just needed to be ported to the network programmability domain. And as expected, such a solution was found: gRPC.

History and Principles of gRPC

Some of the most complex applications shaping the Internet today, such as search engines, social media networks, and cloud infrastructures, are highly distributed by nature. This distribution is a prerequisite to provide the ability to scale and provide a sufficient level of resilience. Modern distributed applications are built using a microservices architecture (see https://microservices.io for more details). The *microservices architecture* basically refers to the splitting of a complex multicomponent application into multiple smaller applications, with each application (called a *microservice*) performing its own small subset of functions. This approach paves the way to simplify each application and remove the dependencies and spaghetti code often seen in monolithic applications, where different parts of the applications are bundled very tightly. On the other hand, in order for an overall application to work, the microservices communicate with each other using remote-procedure calls (RPCs) over a network, as each microservice has an associated IP address and TCP or UDP port. NETCONF is an RPC-based protocol (refer to Chapter 14, "NETCONF and RESTCONF"). So is gRPC. gRPC is a recursive acronym that stands for *gRPC remote-procedure call*. It is also possible to find other interpretations of the *g* part of the name gRPC, such as *general-purpose* or *Google*. Both of those are possible, as Google is the developer and core contributor/maintainer of gRPC. Currently, gRPC is a project within the Cloud Native Computing Foundation (CNCF).

Google made gRPC publicly available in 2015 but had been using the ideas of quick and highly performant RPC to manage the microservices in its data centers since the early 2000s. The name of the protocol back then was Stubby, and it was tightly bundled with Google's service architecture, so it could neither be generalized nor reused by others. (For more details, see https://grpc.io/blog/principles/.) At the same time, in the public space, multiple developments, such as HTTP/2 and SPDY, introduced latency-reducing enhancements and optimization of handling of the requests competing for the same resources. As a result, Google reworked its Stubby protocol into gRPC, leveraging HTTP/2 and its features geared toward enhanced performance (such as binary framing, header compression, and multiplexing; see Chapter 8, "Advanced HTTP," for details) and created an open-source project that was eventually adopted by CNCF and that can be used by a wider audience.

The following concepts form the basis of gRPC:

• **Performance and speed**: One of the core goals of Stubby and, hence, gRPC is to provide fast connectivity between services, and the overall system architecture was developed to implement this concept. One example is the implementation of static paths toward resources rather than dynamic paths such as those implemented by RESTCONF. With a dynamic path, it is possible to include multiple optional queries in the URI, and they need to be parsed before call processing. In contrast, gRPC implements a *static path*, and all the queries must be part of the message body.

• **Microservices oriented**: gRPC was created to interconnect microservices that may be highly distributed across a data center or even between different data centers. It takes into account the networking components of an application, such as delays and losses.

• **Platform agnostic**: gRPC can be used on any platform or operating system, even those that have limited CPU and memory, such as mobile devices and IoT sensors.

• **Open source**: Open-source software is booming now, and for a system to be popular and widely adopted, it is important that its core functionality be open source and free to use. gRPC is open source.

• **Language independent**: gRPC was developed to be available for use in all the programming languages that have wide user bases, such as Python, Go, C/C++, Java, and Ruby. In addition, cross-platform implementation is possible, where the client and server sides are implemented in different languages (for example, a Python client and C++ servers).

• **General purpose**: Because it was built with a focus on microservices and Google architecture, gRPC is generic enough to be used as a communication system between different applications and in different scenarios (for example, the gNMI specification for network management or streaming).

• **Streaming**: gRPC supports various communication patterns, such as basic request/response operations, unidirectional streaming, and bidirectional streaming. It supports both synchronous and asynchronous operations.

• **Payload agnostic**: Originally, gRPC relied on Protocol buffers (discussed in detail later in this chapter) for both data serialization and encoding. Today, it supports any other data encoding, such as JSON or XML. However, Protocol buffers have very dense data encoding and

may provide better efficiency compared to other data encodings.

• **Metadata support**: A lot of applications, especially those communicating over the Internet (which is not a secure environment), require authentication. Application authentication is typically implemented using metadata, which is also the case with gRPC. Generally, gRPC provides the facility to transmit any metadata, which is usually a very useful feature.

• **Flow control**: Network connectivity bandwidth is often unequal inside and outside a data center. For example, the servers inside a data center might be connected with 10 Gbps interfaces, whereas customers connected to the data center from the outside may be connected to low-speed interfaces. gRPC has a built-in mechanism to be able to handle these differences to allow stable connectivity and service operation.

gRPC as a Transport

As you have already seen in this chapter, gRPC is very flexible. gRPC has the following characteristics:

• **No fixed port**: gRPC works over TCP; however, gRPC doesn't have any predefined port. The port is defined solely by the application or vendor. For example, the TCP port that is used for management of network elements via gRPC on Cisco is different from the port used by Arista, which is different from the port used by Nokia. On the one hand, such a flexibility provides an advantage in terms of security (as there are no fixed attack vectors). On the other hand, it makes managing a multivendor network more complicated.

• **No predefined calls and messages**: gRPC is a fast RPC framework. Unlike NETCONF, it doesn't have any predefined structure for its messages. Each application uses its own set of calls and messages, called a *specification*. For example, gNMI is a gRPC specification, as it defines its own set of RPC calls and associated messages.

In a nutshell, gRPC gives you great flexibility to deploy any service you need, with very few limitations. Figure 15-1 provides a high-level overview of the communications flow with gRPC.

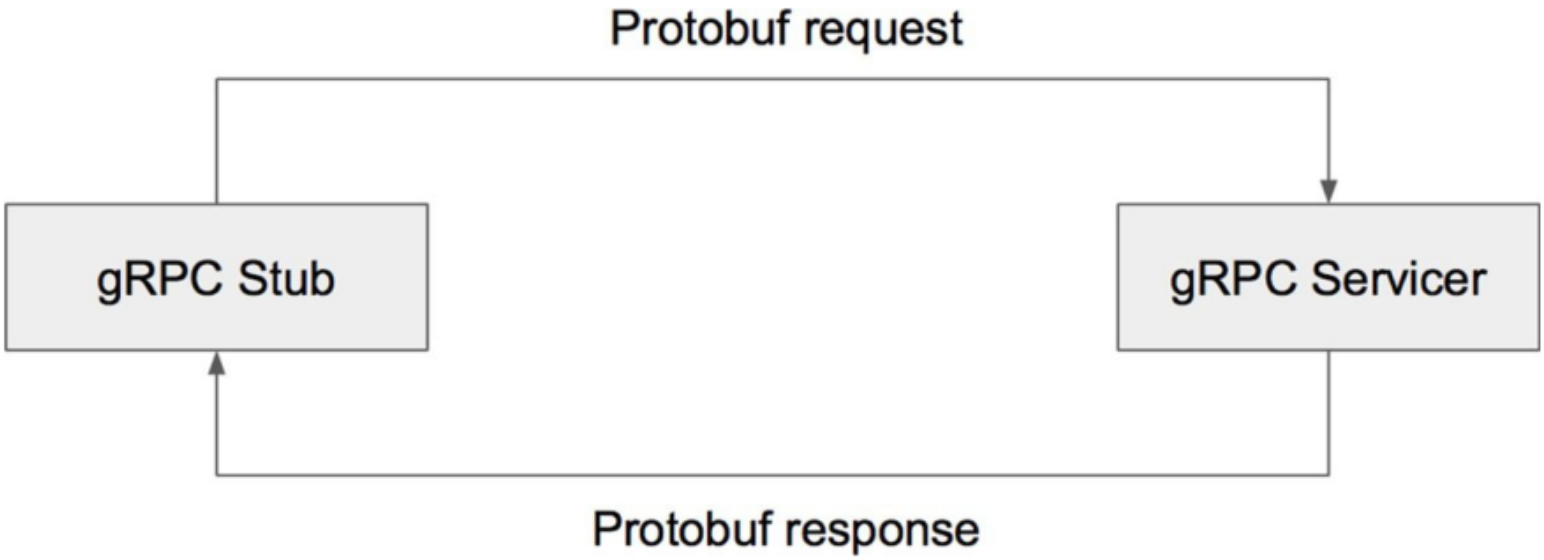

Figure 15-1 *The General gRPC Communications Flow*

In gRPC terminology, *servicer* refers to the server side of the application. Basically, it is the side that listens to customer requests, processes them, and provides responses. The gRPC client side is called *stub*, and it is the side that typically originates the requests and receives the responses from the servicer. The communication between the stub and the servicer is called a *channel*. The channel is specified by the target host address (for example, domain name, IPv4 or IPv6 addresses) and TCP port, and it is established for the duration of the communication and is typically short-lived; however, in some circumstances, it lives for a longer time.

Note

The term *servicer* is a Python-specific term and refers to the interface generated from the service definition. More specifically, a servicer Python class is generated for each service and acts as the superclass of a service implementation. A *function* is generated in the servicer class for each *method* in the service. This will make more sense as you progress through the chapter. The majority of gRPC documentation refers to the two ends of the gRPC communication as stub and server or client and server. To avoid confusion and to keep things simple, the term *server* is replaced by *servicer* throughout the chapter.

In terms of communication patterns, gRPC supports the following scenarios:

• **Unary RPC**: This is one of the simplest communication methods between the stub and the servicer. It involves a single request from the stub to the servicer and a single response back from the servicer to the stub. It is the same as any NETCONF or RESTCONF request/response operation.

• **Server-streaming RPC**: This scenario starts as a unary RPC with the stub's request; however, in the response, the servicer streams a number of messages (sometimes quite a large number of them).

• **Client-streaming RPC**: In this scenario, the stub streams a number of messages to the servicer, and the servicer responds back with a single message.

• **Bidirectional RPC**: Both the stub and the servicer can stream a number of messages to each other. It is important for the streams to be independent of each other so that they can be implemented in an asynchronous manner. The streams may be confirmed by some sort of acknowledgement message from each side.

In addition, gRPC supports transmission of the metadata with each message, pretty much as NETCONF or RESTONF do. One of the popular use cases for metadata is authentication of the messages; this is a mandatory part for the gNMI specification and is based on the gRPC transport.

gRPC is a programming language–neutral technology, which means it can be implemented in virtually any language. It is supported in C++, Go, Ruby, Python, Java, and many other languages. Because gRPC is language independent, the stub and the servicer can be developed and implemented in different languages and interact seamlessly with each other, as long as they follow the same specification. In this book, we focus on Python, and later in this chapter you will see Python scripts to manage network elements using gRPC from the stub's perspective.

One of the key aspects of any protocol or framework used to manage network elements is the set of calls and messages of that protocol. gRPC is very flexible, and it allows you to define your own set of calls and messages. Obviously, to make it work for management of network elements, the messages and RPC calls should be implemented in the network elements' software, which requires access to source code. Later in this chapter, you will learn about gNMI, which is the specification (that is, the set of the calls and messages) used over gRPC transport. But before that, you need to understand protocol buffers, which are discussed next.

The Protocol Buffers Data Format

Google developed protocol buffers (or Protobuf for short) to serve as the main language to define both the gRPC message format and RPC calls. Protocol buffers are one of the core technologies developed and used by Google to serialize data for communication between the elements of highly loaded systems. The reason they are so efficient has to do with the way the data is encoded for transmission: Only key indexes, data types, and values are converted in binary format and sent over the wire. Example 15-1 shows a sample Protobuf message.

Example 15-1 *Simple Protobuf Message*

```
syntax = "proto3";


message DeviceRoutes {

    int32 id = 1;

    string hostname = 2;

    int64 routes_number = 3;

}
```

Example 15-1 consists of two parts: **syntax** and **message**. The **syntax** section defines which version of the protocol buffers are to be used. The most recent and widely used version is Version 3; hence, the **syntax** variable is set to **proto3**. The second section is the **message** section, which is effectively a schema, like a JSON schema or a YANG file, that defines the following:

• **Variables**: The schema defines the names of the variables that may exist in the schema. All variables are optional and may or may not exist in the actual message. In Example 15-1, **id**, **hostname**, and **routes_number** are the names of the variables.

• **Data types**: The schema associates each variable with a certain data type. In Example 15-1, **int32**, **int64**, and **string** are the data types. These data types are built-in types (see https://developers.google.com/protocol-buffers/docs/proto3#json); however, if required, you can create your own data types. For instance, you can create an **enum** type with some options or even use another message defined in the same file as a data type.

• **Indexes**: The schema identifies an index associated with each variable name, as the names aren't included in the Protobuf message sent over the wire. It is the indexes that are included. This is very different from XML and JSON data formats, where the actual key names are transferred. The indexes are both an advantage and a disadvantage of the Protobuf: On the one hand, they allow you to save a lot of bandwidth on the wire, especially in bandwidth-hungry applications such as streaming telemetry. On the other hand, the sender and receiver must have the same schema, or it will be impossible to decode the data out of the binary stream. Indexes must be unique within the level of the message—such as **1**, **2**, and **3** in Example 15-1. In each nested level, though, they can start with **1** again.

With all these details in mind, take a look at the more complicated Protobuf messages in Example 15-2.

Example 15-2 *Complex Protobuf Schema with Multiple Messages and User-Defined Data Types*

```
syntax = "proto3";


enum AddressFamily {

    IPV4 = 0;

    IPV6 = 1;

    VPNV4 = 2;

    VPNV6 = 3;

    L2VPN = 4;

}


enum SubAddressFamily {

  UNICAST = 0;

  MULTICAST = 1;

  EVPN = 2;

}


message Routes {

    AddressFamily afi = 1;

    SubAddressFamily safi = 2;


    message Route {

        string route = 1;

        string next_hop = 2;

    }
```

```
    repeated Route route = 3;
}


message DeviceRoutes {
    int32 id = 1;
    string hostname = 2;
    int64 routes_number = 3;
    Routes routes = 4;
}
```

Although Example 15-2 is much longer than Example 15-1, it strictly follows the guidelines mentioned previously. You can see the named user-defined data types **AddressFamily** and **SubAddressFamily** created using the **enum** (enumerate) built-in data type. Each of the new data types has some allowed values, each associated with an index; the index rules stated previously are applied. These data types in turn are used in the new message **Routes**, where they are associated with the variables **afi** and **safi** and the corresponding indexes **1** and **2**. Inside the message **Routes**, a nested message **Route** is created, and it must be called in the parent message in order to be used. It is called with the index **3** because **1** and **2** are already used by **afi** and **safi**. The message name **Route** is put in the data type position and is prepended by the keyword **repeated**, which means the variable **routes** can be defined several times; this is, effectively, the Protobuf's implementation of lists or arrays.

It is also possible to call one message from another message that is not nested. Hence, you can see in the original **DeviceRoutes** message the new variable **routes**, which has data type **Route** (after the message **Route {}**) and index **4**.

As mentioned earlier in this chapter, protocol buffers are used not only to define the message structure within gRPC but also to identify the structure of the RPC operations: which request message is associated with which operation type and what response message is sent back, as demonstrated in Example 15-3.

Example 15-3 *Sample gRPC Specification in Protobuf*

```
syntax = "proto3";


enum AddressFamily {
    IPV4 = 0;
    IPV6 = 1;
    VPNV4 = 2;
    VPNV6 = 3;
    L2VPN = 4;
}


enum SubAddressFamily {
  UNICAST = 0;
  MULTICAST = 1;
  EVPN2 = 2;
}


message Routes {
    AddressFamily afi = 1;
    SubAddressFamily safi = 2;


    message Route {
        string route = 1;
        string next_hop = 2;
    }
    repeated Route route = 3;
}


message DeviceRoutes {
    int32 id = 1;
    string hostname = 2;
    int64 routes_number = 3;
    Routes routes = 4;
}
```

```
message RouteRequest {

    string hostname = 1;

    AddressFamily afi = 2;

    SubAddressFamily safi = 3;

}


service RouteData {

    rpc CollectRoutes(RouteRequest) returns (DeviceRoutes) {}

}
```

Besides the additional message **RouteRequest**, you can see something completely new in Example 15-3: the **service** part. The **service** part is an abstract definition that ultimately contains the set of **rpc** operations. There should be at least one **rpc** operation per service. In Example 15-3, the **rpc** operation is called **CollectRoutes**, and it states that the client side, which is called stub in gRPC, should send the **RouteRequest** message, whereas the servicer should respond with the **DeviceRoutes** message. This is an example of the definition of *unary RPC*; however, there are three more types, as outlined earlier, and they can be defined in the following manner:

- **Server-streaming RPC**: **rpc CollectRoutes(RouteRequest) returns (*stream* DeviceRoutes) {}**
- **Client-streaming RPC**: **rpc CollectRoutes(*stream* RouteRequest) returns (DeviceRoutes) {}**
- **Bidirectional RPC**: **rpc CollectRoutes(*stream* RouteRequest) returns (*stream* DeviceRoutes) {}**

Altogether, the set of messages and services are named in the specification and stored in a **proto** file. This file is named after its format and has the extension .proto, as shown in Example 15-4.

Example 15-4 *Sample Protocol Buffers Message*

```
$ cat NPAF.proto

syntax = "proto3";


enum AddressFamily {

    IPV4 = 0;

    IPV6 = 1;

    VPNV4 = 2;

    VPNV6 = 3;

    L2VPN = 4;

}

// Further output is truncated for brevity
```

In Example 15-3, the specification is developed with the idea of route distribution between the stub and servicer. Despite the fact that the specification is application dependent, there should be some standard specifications to allow interoperability between the devices in the real world. One of the most popular specifications in the network automation world is gNMI, which is widely used in data centers. You will learn about gNMI at the end of this chapter.

Working with gRPC and Protobuf in Python

Like gRPC, the protocol buffers are programming language independent. This means that protocol buffers can be implemented in any popular programming language, such as C++, Go, Java, or Python. On the other hand, each programming language has its own consumption model that enables the conversion of the **proto** files into the programming language–specific structure. For example, in Python, the data construction is a set of two files with metaclasses that are created as a conversion of the single proto file. There are two files generated out of a single **proto** file because the messages and the service (RPC) part are generated in Python separately.

This conversion is done using a tool developed by Google. This tool, which is called **protoc** (Protocol Buffers Compiler), can create the proper output of the **proto** file in any desired programming language, including Python. There are multiple ways to get **protoc**, but in case of Python, the most sensible way to get it is to install the Python package with the **grpc** tools, as shown in Example 15-5.

Note

All the examples in the rest of the chapter use Python Version 3.7, and backward compatibility with earlier versions isn't guaranteed.

Example 15-5 *Installing **protoc** for Python*

```
$ pip install grpcio grpcio_tools

Collecting grpcio

  Using cached grpcio-1.30.0-cp37-cp37m-macosx_10_9_x86_64.whl (2.8 MB)

Collecting grpcio_tools

  Using cached grpcio_tools-1.30.0-cp37-cp37m-macosx_10_9_x86_64.whl (2.0 MB)

! Some output is truncated for brevity

Successfully installed grpcio-1.30.0 grpcio-tools-1.30.0 protobuf-3.12.2 six-1.15.0
```

As part of the dependency resolution, **grpcio_tools** also installs the **protobuf** package, which is used by **protoc**. At this point, you can convert the **proto** file in the Python metaclasses by using the **protoc** method from **grpc_tools**, as shown in Example 15-6.

Example 15-6 *Converting the **proto** File in Python Metaclasses*

```
$ ls

npaf.proto              requirements.txt        venv


$ python3.7 -m grpc_tools.protoc -I=. --python_out=. --grpc_python_out=. npaf.proto


$ ls

npaf.proto      npaf_pb2.py             npaf_pb2_grpc.py        requirements.txt
```

Despite the fact that the module installed is **grpcio_tools**, in Python it is called **grpc_tools**. (This different naming can be confusing, but the two names refer to the same module.) You need to provide a number of arguments to this module:

• **-I**: Contains the source folder with the **proto** file.

• **--python_out**: Provides the path where the file containing the Python classes for Protobuf messages (ending with _pb2.py) is stored.

• **--grpc_python_out**: Points to the directory where the file containing the Python classes for the gRPC service (ending with _pb2_grpc.py) is located.

As mentioned earlier, each of the resulting files has its own set of associated information. The file with messages is the most complicated, as it contains the conversion of the Protobuf message format in the Python data structure using various descriptors, as demonstrated in Example 15-7.

Example 15-7 *The Auto-generated Python File with Protobuf Messages*

```
$ cat npaf_pb2.py | grep '= _descriptor'

DESCRIPTOR = _descriptor.FileDescriptor(

_ADDRESSFAMILY = _descriptor.EnumDescriptor(

_SUBADDRESSFAMILY = _descriptor.EnumDescriptor(

_ROUTES_ROUTE = _descriptor.Descriptor(

_ROUTES = _descriptor.Descriptor(

_DEVICEROUTES = _descriptor.Descriptor(

_ROUTEREQUEST = _descriptor.Descriptor(

_ROUTEDATA = _descriptor.ServiceDescriptor(
```

As you can see in Example 15-7, the names of the variables are in line with the names of the messages shown in Example 15-3. The file in Example 15-7 is generated automatically and should not be modified manually. The second auto-generated file contains the methods and classes used on the stub and the servicer sides, as you can see in Example 15-8.

Example 15-8 *The Auto-generated Python File with Protobuf Services*

```
$ cat npaf_pb2_grpc.py | grep 'class\|def'

class RouteDataStub(object):

    def __init__(self, channel):

class RouteDataServicer(object):

    def CollectRoutes(self, request, context):

def add_RouteDataServicer_to_server(servicer, server):

class RouteData(object):

    def CollectRoutes(request,
```

The names of the classes are auto-generated from the name of the service—in this case, **RouteData**—and the keyword **Stub** or **Servicer**. As you can imagine, **RouteDataStub** is a class used on the client side, and **RouteDataServicer** is used on the server side, which is listening for the customer requests. **RouteDataServicer** also has the method **CollectRoutes**, which is further defined inside the server-side script to perform any activity necessary based on the logic.

The best way to explain the logic of gRPC in Python is to show the creation of a simple application that has both servicer and stub parts. By now you should be familiar with Python, you should be able to read the code of the gRPC stub provided in Example 15-9.

Example 15-9 *Sample gRPC Client*

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


# Modules

import grpc

from npaf_pb2_grpc import RouteDataStub

from npaf_pb2 import RouteRequest, DeviceRoutes


# Variables

server_data = {

    'address': '127.0.0.1',
```

```
    'port': '51111'
}


# User-defined function
def build_message():
    msg = RouteRequest()
    msg.hostname = 'router1'
    msg.afi = 0
    msg.safi = 0

    return msg


# Body
if __name__ == '__main__':
    with grpc.insecure_channel(f'{server_data["address"]}:{server_data["port"]}') as channel:
        stub = RouteDataStub(channel)
        request_message = build_message()

        print(f'Sending the CollectRequest to {server_data["address"]}:{server_data["port"]}...')
        response_message = stub.CollectRoutes(request_message)

        print(f'Received the response to CollectRequest from {server_data["address"]}:{server_data["port"]}:\n')
        print(response_message)
```

Example 15-9 shows a generic gRPC client that allows you to connect to a gRPC-speaking server. It requires the module called **grpc** (installed in Example 15-5 as **grpcio**) and the building blocks of your Protobuf service and messages:

• **RouteDataStub**: This class is used to send the request **CollectRoutes** to the gRPC servicer and get the response.

• **RouteRequest and DeviceRoutes**: These messages are used to structure and serialize/deserialize messages sent from the stub to the servicer and from the servicer to the stub.

In the **Variables** section of Example 15-9, you can see the simple Python dictionary **server_data**, which contains the IP address and TCP port of the gRPC server. Next is a user-defined function that constructs the data structure using the schema associated with a certain Protobuf message. The data structure, as you can see, is a number of properties of the class **RouteRequest**, which strictly follows the **RouteRequest** message in the original **proto** file: the names of the class's properties are exactly the same as the names of the variables inside the **proto** file (shown in Example 15-3). This data is saved within the function in the object **msg**, which is returned as a result of the function's execution.

In the main part of the application, you can see the object **channel** created using the function **insecure_channel** from the **grpc** module. **insecure** in this function name means that the channel is not protected by encryption (for example, an SSL certificate), and the information is transmitted and received in plaintext. Unlike RESTCONF, gRPC doesn't have an option to skip certificate verification for self-signed certificates. Therefore, you need to think about the certificates' distribution for self-signed certificates or PKI if you want to deploy **secure_channel**; this approach is recommended for production networks. The argument for **insecure_channel** is a string with the IP address and port of the gRPC servicer.

Over the gRPC channel, you need to invoke the stub itself; how it is invoked is application specific. In this case, it is invoked using the class **RouteDataStub**, which is auto-generated out of the gRPC part of the Protobuf specification. From a Python perspective, the stub is also an object that has methods named after the RPC calls in the original **proto** file for this service. The input for the method is a message in the proper format for the **proto** file (**RouteRequest** in this case), and the output is the response message (**DeviceRoute** in this case). As the response is provided, the result of the method execution is saved in the variable **response_message**, which is printed afterward.

Once you have familiarized yourself with Python's gRPC client, you should do the same with the server side; however, to be fair, it is a little bit more complicated, as you can these in Example 15-10.

Example 15-10 *Sample gRPC Server*

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


# Modules
import grpc
from npaf_pb2 import RouteRequest, DeviceRoutes
import npaf_pb2_grpc
from concurrent import futures


# Variables
server_data = {
    'address': '127.0.0.1',
    'port': '51111'
```

```
}


# Classes
class RouteDataServicer(npaf_pb2_grpc.RouteDataServicer):
    def CollectRoutes(self, request, context):
        print(request)
        return self.__constructResponse(request.hostname, request.afi, request.safi)


    def __constructResponse(self, hostname, afi, safi):
        msg = DeviceRoutes()
        msg.hostname = hostname
        msg.id = 1
        msg.routes_number = 10


        msg.routes.afi = afi
        msg.routes.safi = safi


        msg.routes.route.add(route='192.168.1.0/24', next_hop='10.0.0.1')
        msg.routes.route.add(route='192.168.2.0/24', next_hop='10.0.0.1')
        msg.routes.route.add(route='192.168.3.0/24', next_hop='10.0.0.2')


        return msg


# Body
if __name__ == '__main__':
    server = grpc.server(thread_pool=futures.ThreadPoolExecutor(max_workers=10))
    npaf_pb2_grpc.add_RouteDataServicer_to_server(RouteDataServicer(), server)


    print(f'Starting gRPC service at {server_data["port"]}...')
    server.add_insecure_port(f'{server_data["address"]}:{server_data["port"]}')
    server.start()
    server.wait_for_termination()
```

The code in Example 15-10 is a bit more complicated than the code on the client side, mainly because you need to define on the server side what should be done upon receiving the customers' requests. Let's analyze this Python script from the beginning.

To import external artifacts to the server's script, you need the same **grpc** module as in Example 15-9 because it handles the gRPC connectivity. In addition, you need to import the structure of RPC calls, and so the whole auto-generated file **npaf_pb2_grpc** is imported as well, as are the messages used with this operation; therefore, the **RouteRequest** and **DeviceRoutes** classes are imported from the file **npaf_pb2**. Finally, you should also import the **futures** library from the **concurrent** module. gRPC was developed quite recently, with performance and scalability in mind. It therefore needs to be built with multiple resources (that is, with a number of parallel threads processing the calls). Following the import of the external resources, you define the server's IP address and ports that will listen to the gRPC session. The same variable from Example 15-9 matches the one in Example 15-10 to allow the communication between the stub and the servicer.

The next section of Example 15-10 defines a class that controls how the server will behave upon receiving the calls from the customer. This class is based on the auto-generated class **RouteDataServicer** from the imported file **npaf_pb2_grpc**. It is effectively a child of the **npaf_pb2_grpc.RouteDataServicer** class. This approach allows you to focus on only the relevant business logic in your script while benefiting from the session handling that is auto-generated using the **protoc** tool. Within this class, you need to have methods following the names of the RPC operations from the original service for each **proto** file. The method **CollectRoutes** has some external inputs besides its own attributes (**self**): **request** and **context**. The **request** variable contains the message body received from the client, and **context** contains various administrative information (for example, metadata, if used). In Example 15-10, this method just prints the received message and sends back the response.

The response is generated using an additional class that doesn't exist in the original specification. This is a very important concept because nothing prevents you from adding your own methods and attributes according to your requirements. This is why you see the additional private method **__constructResponse** created to build the response message. The response message must have the **DeviceRoutes** format, according to the **proto** file. Therefore, the object **msg** is instantiated from the imported **DeviceRoutes** class. The logic is as follows:

• The properties of the object following the Protobuf message are named after the variables from the **proto** file.

• If there are multiple levels of nesting, variables are stacked and interconnected with the . symbol.

• Whenever you have to deal with an entry defined as *repeated* in the **proto** file, you need to use the function **.add()**, which contains arguments. Those arguments (**route** and **next_hop** in Example 15-3) are defined in the appropriate Protobuf message in the **proto** file.

The **__constructResponse** method returns the **msg** object. Ultimately, this object contains a Protobuf message; hence, the method is called in the original method **CollectRoutes** to generate the reply message.

The last piece of the code actually brings up the gRPC servicer. To do that, the object **server** is instantiated using the **server** class from the

grpc module, where the mandatory argument is the number of workers created to serve the gRPC requests. Afterward, the specific auto-generated function **add_RouteDataServicer_to_server** from **npaf_pb2_grpc** binds the servicer class **RouteDataServicer** you created to the gRPC object **server**. Once this is done, the method **add_insecure_port** associates the TCP socket with the server. The server listens to the incoming requests on this socket. Besides **insecure_port**, there is also a possibility to use **secure_port**, which involves using SSL certificates. At this point,the servicer is ready for operation. It is started, and to allow it to stay for a prolonged period of time, it is put in the mode **wait_for_termination()**, which prevents it from shutting down until it is explicitly terminated by the operator.

By now you should have an understanding of how the basic client and server sides of the gRPC application with your own specification works. So that you can understand it even better, Example 15-11 shows the process of the gRPC server launch.

Example 15-11 *Launching a Sample gRPC Server*

```
$ python npaf_server.py

Starting gRPC service at 51111...
```

The server part is now listening to customer requests, and you can execute the client-side script, as demonstrated in Example 15-12.

Example 15-12 *Testing the Sample gRPC Client*

```
$ python npaf_client.py

Sending the CollectRequest to 127.0.0.1:51111...

Received the response to CollectRequest from 127.0.0.1:51111:


id: 1

hostname: "router1"

routes_number: 10

routes {

  route {

    route: "192.168.1.0/24"

    next_hop: "10.0.0.1"

  }

  route {

    route: "192.168.2.0/24"

    next_hop: "10.0.0.1"

  }

  route {

    route: "192.168.3.0/24"

    next_hop: "10.0.0.2"

  }

}
```

Immediately after the client's script is executed, the gRPC stub sends the **CollectRoute** RPC to the server's address, 127.0.0.1, and port 51111, as defined in the variables. The servicer processes the request and response with the generated message. The server's script contains the **print(request)** instruction, which prints the customer's messages, which are shown in Example 15-13.

Example 15-13 *Logging gRPC Processing at the Servicer*

```
$ python npaf_server.py

Starting gRPC service at 51111...

hostname: "router1"
```

The server's method **CollectRoutes** prints the incoming message, which has the **RouteRequest** format, according to the **proto** file. There is an interesting factor here: If you look at Example 15-3, earlier in this chapter, you can see the three variables (**router**, **afi**, and **safi**), but the printed output has only one (**router**). In fact, you can call these variables as properties inside the script, and they will have value of zero. This is the behavior of **proto3**: If the value of the variable is zero, it is not sent. Always think about efficiency when you work with protocol buffers.

The gNMI Specification

In this chapter, you have learned about gRPC, including the transport mechanisms and how it operates. One of the key things demonstrated earlier in the chapter is that gRPC provides only the transport and type of communication, and the name of the RPCs and format of the messages are application specific. Together, the gRPC services, RPCs, and messages related to a certain application are called the *specification*. gRPC Network Management Interface (gNMI) is a gRPC specification that defines the gRPC service name, RPCs, and messages.

Before we look at the details of the gNMI specification, it is worth understanding what problems it aims to solve and why it has been developed in a particular way.

Originally the only way to perform activities related to management of network elements was to use the CLI and Telnet or SSH. That process is not suitable for collecting operational data, and SNMP was introduced. For decades, SNMP has been a major mechanism for polling the various types of operational states from the network elements, and it has been used in most networks. However, SNMP's ability to change the configuration of the network elements is very limited. Some attempts were made across the industry to build XML-based network management (for example, NX-API in Cisco Nexus), but they were typically limited to particular platforms rather than being widespread.

The next major step in network management was the introduction of YANG and the module-based approach to managing the network elements. The configuration of network elements and their operational data following YANG modules became structured in hierarchical trees consisting of key/value pairs; this approach is much more suitable for management using programmability. The first standard protocol to

support YANG was NETCONF; it standardized the programmable management of network elements across different vendors, and this allowed for API-based interaction with network elements, much as in any other distributed application (for example, communication between a database and a back-end server in a web application). However, the collection of the operational data using NETCONF is a bit complicated, as it must be polled by the server. Some early implementations of NETCONF agents caused very high CPU utilization on routers during requests, which made those agents suitable for configuration changes but not for continuous data polling.

The industry was experimenting with various protocols to effectively distribute operational data in YANG modules—from the network elements to the network management system relying on UDP or TCP protocols with proprietary set of messages and communication flows. As gRPC was developed and became available for the wider public, network vendors added gRPC as another mechanism to distribute or stream the operational data in a process known as *streaming telemetry*.

As had been the case years before, the networking industry had to rely on two protocols to manage network functions in the programmable way. However, this time it was different: gRPC provided a broad framework for various communication types, including the unary type that is suitable for a traditional request/response operation, such as a configuration, and streaming in any direction suitable for the telemetry. The big consumers of the network technologies transitioned to becoming the developers themselves and started looking at how the capability of gRPC could be further applied to network management to find a single protocol that would be suitable for configuration and data collection. This development process took a couple years and resulted in gNMI. To paraphrase the official gNMI specification, gNMI is an attempt to use a single gRPC service definition to cover both configuration and telemetry and to simplify the implementation of an agent on the network devices and allow use of a single network management system (NMS) driver to interact with network devices to configure and collect operational state.

The Anatomy of gNMI

All the details of the gNMI specification are located in the single **proto** file **gnmi.proto**, which you can find in the official OpenConfig/gNMI repository at GitHub (https://github.com/openconfig/gnmi/blob/master/proto/gnmi/gnmi.proto). This file has very good and detailed documentation in the form of internal comments that help you easily understand all the parts of the **proto** file. Example 15-14 shows the structure of the gNMI service and RPCs.

Example 15-14 *gNMI Specification: Service and RPCs*

```
$ cat gnmi.proto

! Some output is truncated for brevity

service gNMI {

  rpc Capabilities(CapabilityRequest) returns (CapabilityResponse);

  rpc Get(GetRequest) returns (GetResponse);

  rpc Set(SetRequest) returns (SetResponse);

  rpc Subscribe(stream SubscribeRequest) returns (stream SubscribeResponse);

}

! Further output is truncated for brevity
```

As you can see in Example 15-14, the gNMI service consists of only one gRPC service and four RPCs, which is fewer RPC types than in NETCONF and fewer API calls than in RESTCONF. However, in terms of functionality, it can perform all the same operations. These are the four gNMI RPCs:

• **Capabilities**: This RPC aims to collect the list of supported capabilities by the network device. This list includes the version of gNMI (the most recent at this writing is 0.7.0) and the supported YANG modules so that the gNMI client knows what can be configured on the target network element or collected from it. From a results perspective, this RPC is similar to the capability exchange that happens during the NETCONF hello process described earlier in this book. This is a unary gRPC operation, which means it has a single client request followed by a single response from the server side.

• **Get**: This RPC implements a mechanism to collect some data from a network device. Depending on the requested scope, the information can be limited by scope (for example, only configuration, only operations, only states), or it is possible to collect all the information available along a certain path that is constructed based on a particular YANG module. Much like **Capabilities**, **Get** is a unary RPC operation. In NETCONF terms, this gRPC operation unites **get-config** and **get** requests.

• **Set**: This RPC is used to change the configuration of the target network device. The scope of configuration change can be quite broad; hence, to clarify it, you can either update the configuration (that is, change the value of a key) along the provided path or replace or delete it completely. Therefore, **Set** is comparable to NETCONF's **edit-config** operation with either the **merge**, **replace**, or **delete** option. Like **Capabilities** and **Get**, **Set** is a unary gRPC operation.

• **Subscribe**: This RPC creates the framework for streaming and event-driven telemetry. It allows the client to signal to the server its interest in receiving information (stream) about the values from a certain path on a regular basis. When the subscription is done, the server starts sending information until the client sends a request to unsubscribe. There is no analogue of this communication type in NETCONF (or in RESTCONF). From the gRPC operation's type standpoint, this is bidirectional streaming.

One of the key concepts in gNMI is **Path**, which is used in **Get**, **Set**, and **Subscribe** RPCs. Effectively, **Path** is very similar to a URI (refer to Chapter 14). Within any application that uses gNMI, the **Path** setting looks as shown in Example 15-15.

Example 15-15 *gNMI Specification: A Sample **Path** Value in the RESTCONF Format*

```
openconfig-interfaces:interfaces/interface[name=Loopback0]
```

The **Path** value starts with the name of the YANG module (in this case, **openconfig-interfaces**) followed by the column separator, :, and then the YANG tree, using the **/** separator for a parent/children relationship. When the list is part of the path, a specific entry from the list is chosen, and the path has an element identification that consists of a key/value pair in square brackets—in this case, **[name=Loopback0]**. In the early days of the gNMI specification, the **Path** value was in the string format; Example 15-15 shows a single value of the string format. However, this is not the case anymore, and in the current version of the gNMI specification, **Path** is serialized following specific Protocol buffer messages. Example 15-16 demonstrates this gNMI format.

Example 15-16 *gNMI Specification: gNMI Path Format*

```
$ cat gnmi.proto

! Some output is truncated for brevity

message Path {

  repeated string element = 1 [deprecated=true];
```

```
  string origin = 2;
  repeated PathElem elem = 3;
  string target = 4;
}

message PathElem {
  string name = 1;
  map<string, string> key = 2;
}
! Further output is truncated for brevity
```

The original string format has been deprecated. A gNMI **Path** value may contain **origin**, which refers to the YANG module in use, and multiple entries (called **elem**s) of the **PathElem** messages. The **PathElem** messages define two elements of **Path**: **name**, which is the name of the relevant YANG leaf, leaf-list, container, or list in string format, and **key**, which is a key/value pair used in the event that it is necessary to specify the element from the list. Using this specification, **Path** is now serialized in the Protobuf binary and looks as shown in Example 15-17.

Example 15-17 *gNMI Specification:* ***Path*** *in the Protobuf Binary*

```
origin: "openconfig-interfaces"
elem {
  name: "interfaces"
}
elem {
  name: "interface"
  key {
    key: "name"
    value: "Loopback0"
  }
}
```

The Get RPC

As we continue to look at the gNMI specification, let's now consider the **Get** RPC. As you saw earlier, in Example 15-14, the **Get** request is defined by the **GetRequest** protocol buffer message, as shown in Example 15-18.

Example 15-18 *gNMI Specification:* ***GetRequest*** *Message*

```
$ cat gnmi.proto
! Some output is truncated for brevity
message GetRequest {
  Path prefix = 1;
  repeated Path path = 2;
  enum DataType {
    ALL = 0;
    CONFIG = 1;
    STATE = 2;
    OPERATIONAL = 3;
  }
  DataType type = 3;
  Encoding encoding = 5;
  repeated ModelData use_models = 6;
  repeated gnmi_ext.Extension extension = 7;
}

message ModelData {
  string name = 1;
  string organization = 2;
  string version = 3;
}
! Further output is truncated for brevity
```

In Example 15-18, the variables **prefix** and **path** are combined together to provide a unique path to the resource that is being polled.

Depending on the gNMI client implementation, it might be that only one of two is provided or that both are provided:

• An empty **prefix** and the **path** value **openconfig-interfaces:interfaces/interface[name=Loopback0]** result in **openconfig-interfaces:interfaces/interface[name=Loopback0]**.

• The **prefix** value **openconfig-interfaces:interfaces** and the **path** value **interface[name=Loopback0]** result in **openconfig-interfaces:interfaces/interface[name=Loopback0]** as well.

• Finally, the **prefix** value **openconfig-interfaces:interfaces/interface[name=Loopback0]** and an empty **path** also result in **openconfig-interfaces:interfaces/interface[name=Loopback0]**.

The reason for this flexibility is that a single **GetRequest** can query multiple resources simultaneously, which allows multiple path entries per message. However, gNMI, like gRPC, is all about efficiency. Therefore, the **prefix** value may contain the part of the path that is common for all the requested resources, whereas the **path** value will have the part that is different. Ultimately, **prefix** is a single entry per message, whereas **path** is a list.

Besides the path, two more elements are mandatory from a logical standpoint: the type, which defines the scope of the information to be collected, and the encoding, which defines the encoding of the information that is expected to be received. The following scopes are available for the requested information:

• **ALL** for all the information available at the provided path.

• **CONFIG** for the read/write elements in the YANG modules used.

• **STATE** for the read-only elements in the YANG modules used.

• **OPERATIONAL** for elements marked in the schema as operational. This refers to data elements whose values relate to the state of processes or interactions running on a device.

In terms of the encoding, although gNMI is defined in the Protobuf, it supports other types of encoding, as shown in Example 15-19.

Example 15-19 *gNMI Specification: Types of Data Encoding*

```
$ cat gnmi.proto
! Some output is truncated for brevity
enum Encoding {
  JSON = 0;
  BYTES = 1;
  PROTO = 2;
  ASCII = 3;
  JSON_IETF = 4;
}
! Further output is truncated for brevity
```

Example 15-19 doesn't mean, though, that the whole message is structured in the binary (**BYTES**) or **JSON** format. It means that the specific part of the response message that contains data extracted from the device will be in one of these formats. The most popular format today, which is implemented across the vast majority of network vendors, is either **JSON** or **JSON_IETF**, which refers to JSON as defined in RFC 7951.

These values are mandatory from a logical standpoint because they are required to identify what you want to collect and how to represent it. However, as you learned earlier, if a key has the value 0, which is also applicable for **enum** data, then it is not sent inside the gRPC/Protobuf. Ultimately, this means that the default encoding is **JSON**, and the default type is **ALL**.

You can use the optional **use_model** variable to further narrow down the specific YANG module by providing the full name, organization, and even version. According to the specification, the **GetRequest** message could look as shown in Example 15-20.

Example 15-20 *gNMI Specification: Sample **GetRequest** Message*

```
path {
  origin: "openconfig-interfaces"
  elem {
    name: "interfaces"
  }
  elem {
    name: "interface"
    key {
      key: "name"
      value: "Ethernet1"
    }
  }
}
path {
  origin: "openconfig-interfaces"
  elem {
    name: "interfaces"
```

```
{\"dhcp-client\": false, \"enabled\": false, \"mtu\": 1500}, \"state\": {\"dhcp-
client\": false, \"enabled\": false, \"mtu\": 1500}}, \"openconfig-if-ip:ipv6\":
{\"config\": {\"dhcp-client\": false, \"enabled\": false, \"mtu\": 1500}, \"state\":
{\"dhcp-client\": false, \"enabled\": false, \"mtu\": 1500}}, \"state\": {\"counters\":
{\"in-broadcast-pkts\": \"0\", \"in-discards\": \"0\", \"in-errors\": \"0\", \"in-fcs-
errors\": \"0\", \"in-multicast-pkts\": \"0\", \"in-octets\": \"0\", \"in-unicast-
pkts\": \"0\", \"out-broadcast-pkts\": \"0\", \"out-discards\": \"0\", \"out-errors\":
\"0\", \"out-multicast-pkts\": \"305\", \"out-octets\": \"29890\", \"out-unicast-pkts\":
\"0\"}, \"description\": \"\", \"enabled\": true, \"index\": 0}}]}}"
    }
  }
}
notification {
}
```

In Example 15-21, you can see that there are two **notification** sections:

- The first one contains information which indicates that the interface named in that section is configured.
- The second one is empty, which means that an interface named in the first section doesn't exist on the device.

Even without looking into the details of the corresponding **GetResponse** message, you can figure out the message structure. However, to prevent you from having any doubts, Example 15-22 shows the relevant protocol buffers messages.

Example 15-22 *gNMI Specification: Messages Associated with the* ***GetResponse*** *Message*

```
$ cat gnmi.proto
! Some output is truncated for brevity
message GetResponse {
  repeated Notification notification = 1;
  Error error = 2 [deprecated=true];
  repeated gnmi_ext.Extension extension = 3;
}

message Notification {
  int64 timestamp = 1;
  Path prefix = 2;
  string alias = 3;
  repeated Update update = 4;
  repeated Path delete = 5;
}

message Update {
  Path path = 1;
  Value value = 2 [deprecated=true];
  TypedValue val = 3;
  uint32 duplicates = 4;
}

message TypedValue {
  oneof value {
    string string_val = 1;
    int64 int_val = 2;
    uint64 uint_val = 3;
    bool bool_val = 4;
    bytes bytes_val = 5;
    float float_val = 6;
    Decimal64 decimal_val = 7;
    ScalarArray leaflist_val = 8;
    google.protobuf.Any any_val = 9;
```

```
    bytes json_val = 10;
    bytes json_ietf_val = 11;
    string ascii_val = 12;
    bytes proto_bytes = 13;
  }
}
! Further output is truncated for brevity
```

At a high level, the **GetResponse** message consists of multiple **notification** entries, which have the format of a **Notification** message. (This is why Example 15-21 shows two **notification** entries.) The **Notification** message is a multipurpose message that is used not only in **GetResponse** but also in **SubscribeResponse** (which is used in streaming). It may contain a **timestamp** setting (depending on whether the network device vendor has implemented it), which provides the time, in nanoseconds, from the beginning of the epoch (January 1, 1970, 00:00:00 UTC). It may also include **prefix**, explained earlier, and **alias**, which may be used if supported by the network device to compress the path; when an alias is created for a path, the messages contains the value of the alias instead of a prefix, which helps reduce the length of the transmitted message. However, all those variables are optional. What is mandatory is either the **update** or **delete** variable; **update** is used to provide the information per **GetResponse** if the information exists, and **delete** contains the paths of the elements that were deleted (**delete** is not used in the current version of the specification for GetRequest and is used for SetRequest only).

Finally, the **Update** message contains the path to the resource being polled and the value stored in the variable **val**, which is further defined by the message **TypedValue**. Inside this message is the value of **val**, which is a long list of the data types. (You might notice in Example 15-21 that the response is in the **json_ietf_val** data type.)

The Set RPC

The **Set** RPC operation is generally similar to the **Get**, as it is also a unary gRPC type; however, it has its own set of messages. Example 15-23 shows the details of the **SetRequest** message.

Example 15-23 *gNMI Specification:* ***SetRequest*** *Message Format*

```
$ cat gnmi.proto
! Some output is truncated for brevity
message SetRequest {
  Path prefix = 1;
  repeated Path delete = 2;
  repeated Update replace = 3;
  repeated Update update = 4;
  repeated gnmi_ext.Extension extension = 5;
}
! Further output is truncated for brevity
```

Although the message in Example 15-23 is very compact, it allows you to choose the necessary operation—and it is possible to have multiple operations in a single message due to **repeated** keyword. These are the potential operations:

• **prefix**: This operation is covered earlier in this chapter, in the section "The **Get** RPC."

• **delete**: This operation contains the paths that are to be removed. You use this operation when you delete a part of the configuration on a network device.

• **replace**: This operation contains the data in the **Update** message format (refer to Example 15-22). Because the **Update** message contains the path toward the resource and **val** with the actual data to be set along the path, the logic of this operation causes the data provide to overwrite what already exists in the destination node.

• **update**: This operation also relies on the **Update** message format. However, the logic of the operation is different: The data from the **val** entry is merged with what already exists in the node that has the provided path. If nothing exists, then a new entry is created.

Example 15-21 shows a **GetRequest** for the Loopback0 interface; however, in that example, **GetRequest** is not configured on the target network function. Hence, in the **GetResponse** from Example 15-22, the notification for this request is empty. Example 15-24 illustrates the use of **SetRequest** to configure the Loopback0 interface at the target network function.

Example 15-24 *gNMI Specification: Sample* ***SetRequest*** *Message*

```
update {
  path {
    origin: "openconfig-interfaces"
    elem {
      name: "interfaces"
    }
    elem {
      name: "interface"
      key {
        key: "name"
        value: "Loopback0"
```

```
  Operation op = 4;
}
! Further output is truncated for brevity
```

The parent message **GetResponse** contains **prefix**, **timestamp**, and **response**, formatted according to the **UpdateResult** message. In the current version of the gNMI specification (0.7.0), the **response** message contains the path of the affected resource and the result of the operation in the variable **op**, which corresponds to the **SetRequest** type: **DELETE**, **REPLACE**, or **UPDATE**.

The Capabilities RPC

You have learned details about some complicated RPCs and their associated messages. The **Capabilities** RPC is significantly easier to understand. As mentioned earlier, the **Capabilities** RPC is used to collect information about the YANG modules supported by the target network device. The **CapabilityRequest** message in Example 15-27 is the simplest Protobuf message described in this book.

Example 15-27 *gNMI Specification:* ***CapabilityRequest*** *Message Format*

```
$ cat gnmi.proto
! Some output is truncated for brevity
message CapabilityRequest {
}
! Further output is truncated for brevity
```

This message effectively doesn't have a variable at all. In fact, this is quite logical: When the client queries a network device about what capabilities it supports, it doesn't need to specify anything else. Example 15-28 shows the format of the **CapabilityResponse** message.

Example 15-28 *gNMI Specification:* ***CapabilityResponse*** *Message Format*

```
$ cat gnmi.proto
! Some output is truncated for brevity
message CapabilityResponse {
  repeated ModelData supported_models = 1;
  repeated Encoding supported_encodings = 2;
  string gNMI_version = 3;
}
! Further output is truncated for brevity
```

The **ModelData** and **Encoding** message formats (which are explained in the section "The **Get** RPC," earlier in this chapter) are provided with the keyword **repeated** because a network device might (and typically does) support multiple YANG modules or encoding formats. The variable **gNMI_version** is not repeated, though, as the network device would have a single version of the gNMI specification implemented.

A sample **CapabilityRequest** message is not shown, as this type of message doesn't contain any data. However, a **CapabilityResponse** message can be rather interesting, as shown in Example 15-29.

Example 15-29 *gNMI Specification: Sample* ***CapabilityResponse*** *Message*

```
supported_models {
  name: "openconfig-packet-match"
  organization: "OpenConfig working group"
  version: "1.1.1"
}
supported_models {
  name: "openconfig-hercules-platform"
  organization: "OpenConfig Hercules Working Group"
  version: "0.2.0"
}
supported_models {
  name: "openconfig-bgp"
  organization: "OpenConfig working group"
  version: "6.0.0"
}
!
! Some output is truncated for brevity
!
supported_encodings: JSON
supported_encodings: JSON_IETF
supported_encodings: ASCII
gNMI_version: "0.7.0"
```

```
  Encoding encoding = 8;
  bool updates_only = 9;
}

message Subscription {
  Path path = 1;
  SubscriptionMode mode = 2;
  uint64 sample_interval = 3;
  bool suppress_redundant = 4;
  uint64 heartbeat_interval = 5;
}

enum SubscriptionMode {
  TARGET_DEFINED = 0;
  ON_CHANGE      = 1;
  SAMPLE         = 2;
}

message QOSMarking {
  uint32 marking = 1;
}
! Further output is truncated for brevity
```

SubscribeRequest consists of one of the three possible types: **SubscribeList**, **Poll**, or **AliasList**. **AliasList** is used to create a mapping of the paths to short names to improve the efficiency of the communication by reducing the data sent between the gNMI client (the NMS) and the gNMI server (the network device). The aliases are optional. The **Poll** message, as you can see in Example 15-31, doesn't have any further parameters. In fact, it is used in poll-based information collection, where the client sends the **Poll** messages to trigger the network device to send the update when the subscription is created.

The subscription is created inside the **SubscribeList** message, which has the following fields:

• **prefix**: This field contains the part of the path that is common for all the subscriptions.

• **Subscription**: This field (or fields) contains a path that it wants to receive the information from, and a subscription mode (defined by the target, on information change, and sampling over a certain time interval). In addition, **sample_interval** identifies how often updates are sent from the server to the client, **suppress_redundant** optimizes the transmission in the update in such a way that the updates aren't sent if there are no changes even if the sample time comes, and **heartbeat_interval** changes the behavior of **suppress_redundant** so that the update is sent once per **sample_interval** to notify the client that the server is up and running.

• **use_aliases**: This field notifies the server about whether the aliases should be used.

• **qos**: This field tells the server which QoS marking it should use for the telemetry packets.

• **mode**: This field sets the type of the subscriptions for all the paths. The available options are **STREAM** (which means the server streams the data on the defined **sample_interval**), **POLL** (which means the server sends the updates per the client's **Poll** message), and **ONCE** (which means the server sends the response once per the client's request and then terminates the gRPC session).

• **allow_aggregation**: This field checks whether there are any data paths available for aggregation so that it can aggregate them into a single update message and send a bulk update to the client.

• **use_models**: This field makes it possible to narrow the search for the paths specified in **Subscription** to certain data models.

• **encoding**: This field sets the data format that the server should use to send updates to the client.

• **updates_only**: This field changes the way the information from the server is sent; instead of sending the state of the identified paths, it sends only updates to the states compared to the previous information distribution.

Sample messages showing the **SubscribeRequest** and the **SubscribeResponse** messages are omitted for brevity, but you can guess their content based on the examples of other messages in this chapter.

Note

In some of the messages shown in this chapter, you might have noticed references to the gNMI extensions. These extensions are available for future possible development of gNMI functionality and are optional to any of the messages. They aren't widely used today.

Managing Network Elements with gNMI/gRPC

Earlier in this chapter, you saw Python code that can be used to create the client side (the stub) and the server side (the servicer) of a gRPC-based application. In the case of gNMI, the servicer is already deployed on a network device. Therefore, you just need to enable it either in secure mode (using the SSL certificates and/or PKI) or insecure mode (without encryption) and create the code for the stub part. Depending on the network operating system, some of the modes (such as insecure) may be not available.

Before you enable gNMI, you need to ensure that your software supports it. If it does, you can enable gNMI on Cisco IOS XE network devices as shown in Example 15-32.

Example 15-32 *Enabling gNMI in Insecure Mode in Cisco IOS XE*

```
NPAF_R1> enable

NPAF_R1# configure terminal

NPAF_R1(config)# gnmi-yang

NPAF_R1(config)# gnmi-yang server

NPAF_R1(config)# gnmi-yang port 57400

NPAF_R1(config)# end
```

gNMI doesn't have a predefined port number; hence, different vendors implement different default values for the gNMI service. You can set this parameter to the value you like. However, if you don't set the value, Cisco IOS XE uses the default, which is TCP/50052. Refer to the official Cisco documentation to enable the secure gNMI server on Cisco IOS XE network devices. You can verify whether the service is operational as shown in Example 15-33.

Example 15-33 *Verifying That gNMI Is Operational in Cisco IOS XE*

```
NPAF_R1# show gnmi-yang state


State Status

--------------------------------

Enabled Up
```

Besides Cisco IOS XE, gNMI is supported in the latest releases of Cisco IOS XR (starting from 6.0.0) and Cisco NX OS (starting from 7.0(3)I5(1)) as well. Moreover, the vast majority of other network operating systems, such as Nokia SR OS, Arista EOS, and Juniper Junos, support gNMI as well. Therefore, you can use it as the main protocol to manage your network.

There are currently no Python libraries that work well with gNMI specifically. However, the library **grpc** (used earlier in this chapter) can help you create the proper Python scripts. If you cloned the gNMI specification from the official repository, you have the gnmi.proto file, and by using the **grpc_tools.protoc** module, you can convert the file into a set of Python metaclasses as shown in Example 15-34.

Example 15-34 *Creating Python Metaclasses for the gNMI Specification*

```
$ ls

gnmi.proto


$ python -m grpc_tools.protoc -I=. --python_out=. --grpc_python_out=. Gnmi.proto

$ ls

gnmi_pb2_grpc.py  gnmi_pb2.py  gnmi.proto


State Status

--------------------------------

Enabled Up
```

Earlier in this chapter, in Example 15-9, you saw how to create the gNMI stub for an arbitrary specification. In this section, you will create it by using the gNMI specification, leveraging explanations provided earlier in this chapter. Example 15-35 shows a script to use the **Get** RPC from the gNMI specification.

Example 15-35 *Sample gNMI **Get** Python Script*

```
$ cat gNMI_Client.py

#!/usr/bin/env python


# Modules

import grpc

from gnmi_pb2_grpc import *

from gnmi_pb2 import *

import re

import sys

import json


# Variables

path = {'inventory': 'inventory/inventory.json', 'network_functions': 'inventory/network_functions'}


# User-defined functions

def json_to_dict(path):

    with open(path, 'r') as f:

        return json.loads(f.read())
```

```
def gnmi_path_generator(path_in_question):
    gnmi_path = Path()
    keys = []

    # Subtracting all the keys from the elements and storing them separately
    while re.match('.*?\[.+?=.+?\].*?', path_in_question):
        temp_key, temp_value = re.sub('.*?\[(.+?)\].*?', '\g<1>', path_in_question).split('=')
        keys.append({temp_key: temp_value})
        path_in_question = re.sub('(.*?\[).+?(\].*?)', f'\g<1>{len(keys) - 1}\g<2>', path_in_question)

    path_elements = path_in_question.split('/')

    for pe_entry in path_elements:
        if not re.match('.+?:.+?', pe_entry) and len(path_elements) == 1:
            sys.exit(f'You haven\'t specified either YANG module or the top-level container in \'{pe_entry}\'.')

        elif re.match('.+?:.+?', pe_entry):
            gnmi_path.origin = pe_entry.split(':')[0]
            gnmi_path.elem.add(name=pe_entry.split(':')[1])

        elif re.match('.+?\[\d+?\].*?', pe_entry):
            key_id = int(re.sub('.+?\[(\d+?)\].*?', '\g<1>', pe_entry))
            gnmi_path.elem.add(name=pe_entry.split('[')[0], key=keys[key_id])

        else:
            gnmi_path.elem.add(name=pe_entry)

    return gnmi_path


# Body
if __name__ == '__main__':
    inventory = json_to_dict(path['inventory'])

    for td_entry in inventory['network_functions']:
        print(f'Getting data from {td_entry["ip_address"]} over gNMI...\n\n')

        metadata = [('username', td_entry['username']), ('password', td_entry['password'])]

        channel = grpc.insecure_channel(f'{td_entry["ip_address"]}:{td_entry["port"]}', metadata)
        grpc.channel_ready_future(channel).result(timeout=5)

        stub = gNMIStub(channel)

        device_data = json_to_dict(f'{path["network_functions"]}/{td_entry["hostname"]}.json')

        gnmi_message = []
        for itc_entry in device_data['intent_config']:
            intent_path = gnmi_path_generator(itc_entry['path'])
            gnmi_message.append(intent_path)

        gnmi_message_request = GetRequest(path=gnmi_message, type=0, encoding=0)
        gnmi_message_response = stub.Get(gnmi_message_request, metadata=metadata)
```

```
print(gnmi_message_response)
```

The script shown in Example 15-35 was used to generate the **GetRequest** and **GetResponse** messages shown earlier in this chapter.

Summary

This chapter describes gRPC transport, Protobuf data encoding, and gNMI specification, including the following points:

• gRPC is a fast and robust framework created by Google to allow communication between applications.

• gRPC supports four types of communication: unary RPC, server-based streaming, client-based streaming, and bidirectional streaming.

• gRPC doesn't define any message types or formats. Each application may have its own set of messages and RPC operations.

• Protocol buffers are used to specify the services, the RPCs, and the message formats for the gRPC services.

• Protocol buffers are effectively schemas that define for each entry the data type, the name of the key, and the index that is used to identify the key upon serialization.

• By default, Protobuf uses binary serialization; however, it may support other types, such as JSON or ASCII text.

• The set of services, RPCs, and messages defined in Protobuf format for a specific gRPC service is called a specification. gNMI is a gRPC specification developed to manage network elements.

• gNMI covers network management both in terms of interacting with the network elements using the unary operation (request/response) suitable for configuration and using a bidirectional streaming operation that is suitable for collecting the live operational states.

Chapter 16. Service Provider Programmability

This chapter covers a special use case for network programmability and automation: service provider network programmability. This domain poses some unique challenges due to the special nature of typical service provider networks in terms of size, scalability, and the diverse types of interfaces and traffic that these networks are expected to transport. Service provider networks are also bound by very stringent KPIs due to the probable revenue loss associated with any downtime. These challenges have recently been amplified due to the introduction and adoption of the 5G technology. This chapter begins with a look at the software-defined networking (SDN) framework.

The SDN Framework for Service Providers

A number of challenges arise in service provider networks due to new application consumption patterns and associated development of new technologies and architectures.

Requirements for Service Provider Networks of the Future

At this writing, international standard bodies define 5G as the latest generation of mobile communication. Besides providing the fastest ever mobile broadband access to the Internet, 5G also introduces a radically new approach to mobile networking called *network slicing*. In a nutshell, a network slice is dedicated set of resources (for example, on the radio interface between a mobile phone and a cellular base station, on router buffers, on router-to-router links, on the mobile packet core), as shown on Figure 16-1.

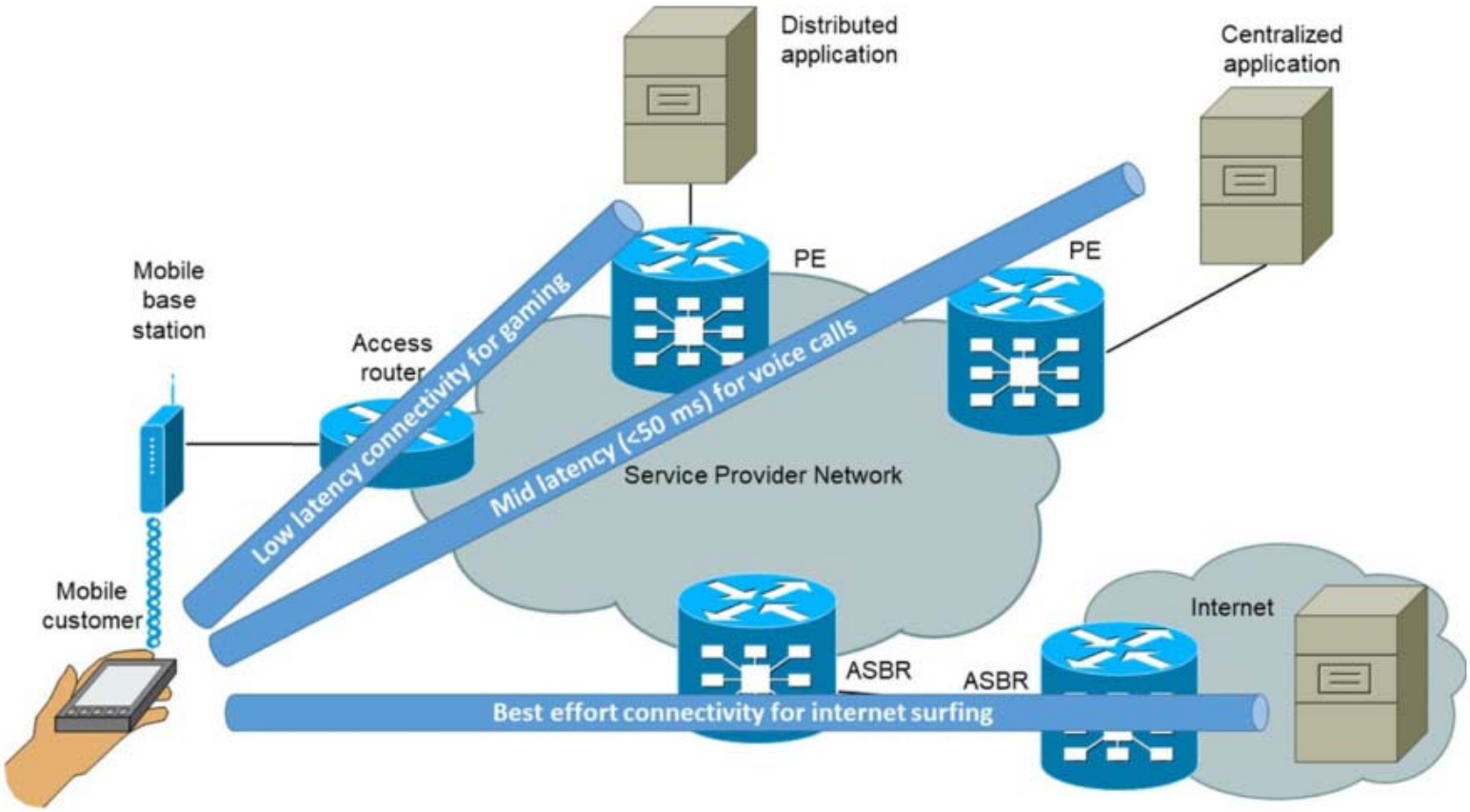

Figure 16-1 *Network Slicing*

3GPP 5G requirements calls for network slicing support for various use cases, such as enhanced mobile broadband (eMBB), massive machine-type communication (mMTC), and ultra-reliable and low latency communications (URLLC). You may think that such requirements are quite trivial in the sense that QoS configuration or Multiprotocol Label Switching Traffic Engineering (MPLS-TE) has been implemented in service provider networks for ages. That is true, but network slicing has one very significant difference from traditional QoS and MPLS-TE configuration: A network slice must be created dynamically when you launch an application on your mobile phone and must be released when you switch off the application. Obviously, such a process can't exist in the traditional service provider world, where all the network changes are reviewed by change advisory boards and implemented manually (through the CLI or a GUI) by network engineers. Network slicing requires full network programmability and an integrated feedback loop so that the network configuration required to create and release the slice is done programmatically, requiring no human intervention at any stage.

The fact that service provider networks are geographically distributed adds even more complexity to the case just described. 5G network slices set a number of requirements for end-to-end KPIs regarding guaranteed bandwidth, latency, packet drops, and jitter. To fulfill those requirements, a service provider network must be able to collect a massive amount of related telemetry information, such as per-link latency, QoS buffer utilization, and actual routing topology. Then network elements must export all that information to a centralized entity, often called an SDN controller, which in turn must be capable of analyzing this information and automatically making decisions based on preprogrammed policies or API calls. Given the multidomain character of 5G networking, SDN controller hierarchy would be is required to abstract the complexity by keeping detailed information within domain-specific controllers while using a multidomain controller to create an end-to-end network slice. The IETF's Traffic Engineering Architecture and Signaling (TEAS) working group has produced RFC 8453, "Framework for Abstraction and Control of TE Networks (ACTN)," which addresses this type of traffic engineering.

The case just described illustrates the direction in which service providers need to transform their networks and operational processes to keep pace with developments in business and society. This chapter explains the architecture, technology, and protocols required to achieve automation via programmability.

SDN Controllers for Service Provider Networks

It's important to understand what SDN means in the context of this chapter as SDN is commonly discussed today for solutions in different industries that solve different problems. In this chapter, we split SDN into overlay and underlay categories.

Overlay SDN covers all kinds of SDN technologies that don't affect the path of the traffic through the network and focus only on encapsulation of the traffic on an edge router (customer premises equipment [CPE] for service providers or a leaf switch in a data centers) and decapsulation

on another edge router. The most popular technologies are MPLS over GRE (or MPLS over UDP),VXLAN, and GENEVE. All these technologies encapsulate incoming customer traffic and then forward it over the network using standard routing mechanisms, following standard IGP/BGP path selection. For signaling of the customer routes, which might be IP prefixes, MAC addresses, or a mixture of the two, one of the most popular and useful protocols is BGP-EVPN. For the enterprise segment, the solution class is called SD-WAN, and Cisco offers Meraki and Viptela as products. In the data center field, Cisco offers ACI (Application Centric Infrastructure) as an SDN solution. Other vendors also offer solutions in this area, such as Nokia Nuage and Juniper Contrail, which are applicable both for data centers and SD-WAN deployments.

Note

Although overlay SDN is valuable, it is beyond the scope of this book, so we don't discuss it further.

In contrast to overlay SDN, *underlay SDN* directly influences the path of traffic between edge routers through the use of traffic engineering technologies. It can be both traditional MPLS-TE using RSVP for path signaling or modern flavors of Segment Routing Traffic Engineering (SR-TE), which is more suitable for network programming. The real power of traffic engineering is unleashed when an SDN controller has an end-to-end view of the network and can signal a proper path (that is, a path that meets the constraints) based on this end-to-end view. To collect end-to-end topology, an SDN controller uses BGP-LS, which involves a new specific address family and distributes IS-IS or OSPF topology to the SDN controller. To signal MPLS label-switched paths (LSPs) down to edge routers, the SDN controller uses PCEP (Path Computation Element Protocol), which is defined in RFC 5450. Figure 16-2 shows these protocols working together to provide a programmable network path.

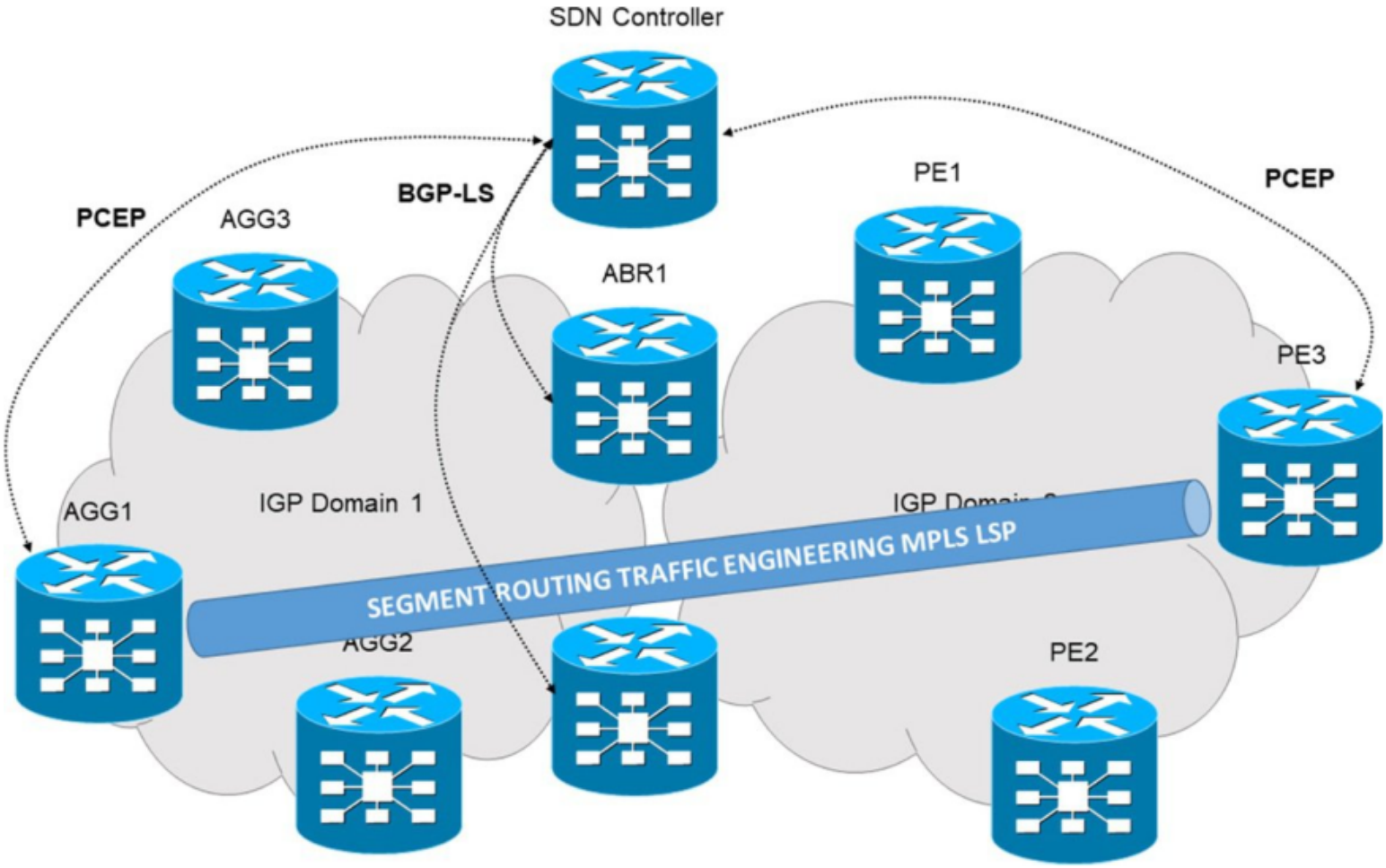

Figure 16-2 *Underlay SDN Protocol Suite*

The underlay solution works as follows:

1. The network runs a link-state IGP (IS-IS or OSPF), which is typical for service provider networks, with Segment Routing (SR) enabled. (SR is the most modern mechanism developed to perform MPLS in the network.) The original mechanism, MPLS-TE, which is a framework that allows the edge router to predefine the exact path through the network by signaling it across the routers using RSVP-TE, is generally supported as well; however, MPLS-TE is not a primary focus of this book, as it isn't primarily for service provider networks or automation. When enabled, SR ensures that there are per-node and per-link MPLS labels available.

2. Some network functions (basically, the routers that have the best view of the network, such as ABRs) establish BGP-LS sessions with the SDN controller.

3. These network functions export their link-state database created using an IGP (OSPF or IS-IS) to the SDN controller so that the SDN controller has an end-to-end view of the network topology, including routing and MPLS labels.

4. Ingress/egress network functions peer with SDN controllers using PCEP in order to get proper end-to-end MPLS LSPs.

5. Upon request, the SDN controller pushes the corresponding Segment Routing label stack (or Explicit Route Object - ERO), which is a list of the next hops (that is, IP addresses of the network functions) the MPLS-TE tunnel should take through the network to the ingress router to instantiate the MPLS LSPs.

These steps provide a rather simplified view of the process that occurs in the SDN-controlled network, but they highlight the structure of the process. The following sections describe the parts of the process in greater detail.

Segment Routing (SR)

The first major building block in service provider programmability and SDN is Segment Routing (SR). This chapter describes basic SR

concepts, such as how it works on its own and with SDN controllers.

Note

To learn more details about SR, you should read *Segment Routing, Part I* and *Segment Routing, Part II* by Clarence Filsfils, who has significantly contributed to the development of SR.

Segment Routing Basics

The SR technology is described in RFC 8402, "Segment Routing Architecture," which was published in July 2018. Work on SR started some time ago, and router code has supported SR support for a while. Several big networks (for instance, Vodafone Germany) have been running SR for a couple years already. Figure 16-3 shows a list of IETF activities; you can see that only a few documents are standardized, and work is ongoing to extend SR capabilities.

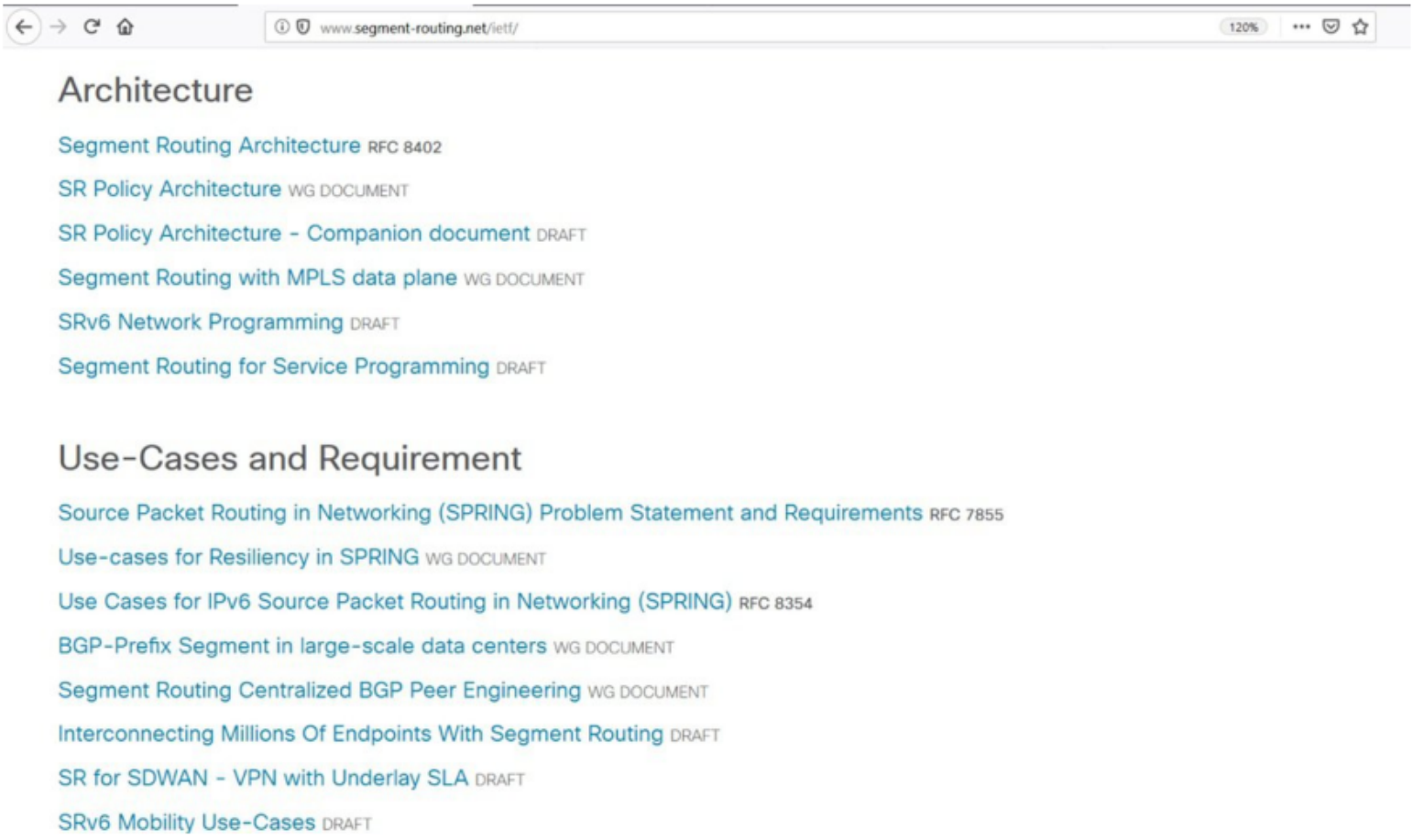

Figure 16-3 *Segment Routing IETF Drafts*

This chapter describes Segment Routing with the MPLS data plane—referred to as *SR-MPLS*. It is also possible to use Segment Routing with the IPv6 data plane, where information is encoded in IPv6 address; this is referred to as SRv6. However, SRv6 is beyond the scope of this book.

What makes SR-MPLS the best choice for network programmability compared to other MPLS data plane protocols? Basically, SR implements a source routing paradigm. *Source routing* means that the head-end router (or ingress router in the MPLS domain) defines the full end-to-end path from itself to the tail-end route (or egress router in the MPLS domain). Once the head-end router has defined the path, the transit routers (typically called label switch routers [LSRs]) perform forwarding according to the instructions encoded in the packets. In the case of SR, MPLS labels are these instructions.

Before we can get into how SR works, it is important to understand some core definitions related to SR:

• **Segment**: This is a general instruction to at router that describes what to do with a packet. In the most common scenario, the segment value is represented as the MPLS label toward the egress router. In more sophisticated cases, segments can define egress interfaces, service chains, QoS parameters, and so on.

• **SRGB (Segment Routing Global Block)**: This is a range of labels used for Segment Routing globally, which means this information is distributed across the whole SR-MPLS domain and must be consistent across this domain. For Cisco IOS XR, the default value is the range 16000 to 23999; other network operating systems (such as Juniper Junos OS, Arista EOS, and Nokia SR OS) have different default ranges.

• **SRLB (Segment Routing Local Block)**: This is a range of labels used locally to each router within an SR-MPLS domain, and these labels are coming from a range other than SRGB. In Cisco IOS XR, there is no specific range for this, and the labels are assigned from the range starting from 24000—which means the range is shared with LDP, MPLS-TE, and BGP. Each Adj-SID (described later in this list) becomes the label from this range. In contrast to SRGB, these labels have local meaning. Therefore, there might be duplications of these labels within an SR-MPLS domain. Although you might think that this would lead to traffic drops, this is not the case (and you will learn why shortly).

• **Prefix-SID (Prefix Segment Identifier)**: This is a segment associated with any prefix within the IGP domain. It doesn't seem necessary for this prefix to be local to the router that generates this Prefix-SID. It can be represented as an index value that is used in calculating the SR-MPLS label: label = SRGB starting label + index (for example, if the index is 11, the SR-MPLS label is 16000 + 11 = 16011). It can also be represented as an absolute label value out of the SRGB range. This parameter can be configured for each interface.

• **Node-SID (Node Segment Identifier)**: This is a particular case of the Prefix-SID (with the flag *N* set) that typically represents the router itself and is usually configured at Loopback0 or any other interface that is associated with the router ID.

• **Adj-SID (Adjacency Segment Identifier)**: This segment identifies unidirectional IGP adjacency (that is, a neighbor in OSPF or IS-IS).

As you examine the examples on the following pages, you will see these concepts in action and become more familiar with the ideas of Figure 16-4 shows the lab topology that is used for the following examples.

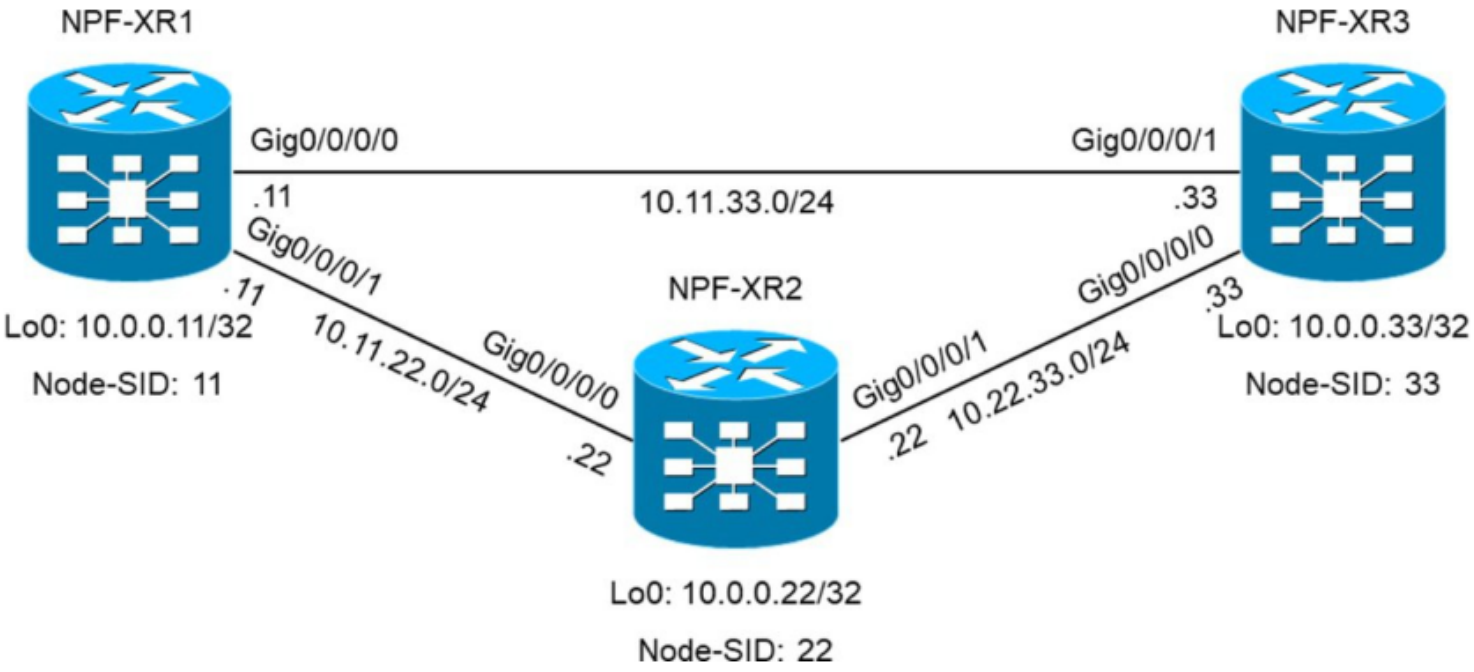

Figure 16-4 *Segment Routing Lab Topology*

This topology is quite simple. The one part you may not be familiar with is Node-SID, which, as described above, represents a router in the SR-MPLS domain and is associated with Loopback0 interfaces. Example 16-1 shows how to enable Segment Routing in this network, which is running IS-IS IGP. (OSPF is also supported.)

Example 16-1 *Configuring Segment Routing in a Network Running IS-IS*

```
RP/0/0/CPU0:NPF-XR1(config)#show conf
Sun Mar 31 18:29:09.914 UTC
Building configuration...
!! IOS XR Configuration 6.5.1
router isis CORE
 is-type level-2-only
 net 49.0000.0100.0000.0011.00
 log adjacency changes
 address-family ipv4 unicast
  metric-style wide
  advertise passive-only
  segment-routing mpls sr-prefer
 !
 interface Loopback0
  passive
  address-family ipv4 unicast
   prefix-sid index 11
  !
 !
 interface GigabitEthernet0/0/0/0
  point-to-point
  address-family ipv4 unicast
  !
 !
 interface GigabitEthernet0/0/0/1
  point-to-point
  address-family ipv4 unicast
  !
 !
!
end
```

Note

If you are not familiar with IS-IS configuration and operation, see *Routing TCP/IP, Volume I* by Jeff Doyle.

Example 16-1 shows an IS-IS configuration that is quite standard for service provider networks. Note in this example that you need to enable **metric-style wide** to allow IS-IS to propagate Segment Routing information. Otherwise, the configuration of SR is straightforward: You need to enable it within the appropriate address family by using the **segment-routing mpls sr-prefer** command, where **segment-routing mpls** enables SR for IS-IS generally, and the keyword **sr-prefer** instructs the router to prefer SR labels to LDP by changing the protocol's AD (administrative distance). After that, you configure the Node-SID by issuing the command **prefix-sid index 11** within the interface/address-family context. With the keyword **index** you do not provide an absolute value but a local ID, and each router within the SR-MPLS domain calculates the SR label value on its own by using SRGB and this ID. (SRGB is not explicitly configured in this example because it is implicitly enabled with the default value 16000-23999, as mentioned earlier.)

Example 16-2 shows the output of the IS-IS information (LSDB), including Segment Routing values generated by NPF-XR1.

Example 16-2 *IS-IS Link State Information for NPF-XR1*

```
RP/0/0/CPU0:NPF-XR1#show isis database verbose NPF-XR1.00-00

Sun Mar 31 19:00:38.625 UTC


IS-IS CORE (Level-2) Link State Database

LSPID                 LSP Seq Num  LSP Checksum  LSP Holdtime/Rcvd  ATT/P/OL

NPF-XR1.00-00       * 0x00000006   0x69ce        380  /*            0/0/0

  Area Address:   49.0000

  NLPID:          0xcc

  IP Address:     10.0.0.11

  Hostname:       NPF-XR1

  Metric: 10         IS-Extended NPF-XR2.00

    Interface IP Address: 10.11.22.11

    Neighbor IP Address: 10.11.22.22

    Link Maximum SID Depth:

      Subtype: 1, Value: 10

    ADJ-SID: F:0 B:0 V:1 L:1 S:0 P:0 weight:0 Adjacency-sid:24001

  Metric: 10         IS-Extended NPF-XR3.00

    Interface IP Address: 10.11.33.11

    Neighbor IP Address: 10.11.33.33

    Link Maximum SID Depth:

      Subtype: 1, Value: 10

    ADJ-SID: F:0 B:0 V:1 L:1 S:0 P:0 weight:0 Adjacency-sid:24003

  Metric: 0          IP-Extended 10.0.0.11/32

    Prefix-SID Index: 11, Algorithm:0, R:0 N:1 P:0 E:0 V:0 L:0

    Prefix Attribute Flags: X:0 R:0 N:1

  Router Cap:     10.0.0.11, D:0, S:0

    Segment Routing: I:1 V:0, SRGB Base: 16000 Range: 8000

    SR Algorithm:

      Algorithm: 0

      Algorithm: 1

    Node Maximum SID Depth:

      Subtype: 1, Value: 10
```

The core SR components, which are shaded in Example 16-2, are as follows:

- **Adj-SIDs**: There are two of these, each associated with an IS-IS neighbor.
- **Prefix-SID**: This is configured as an index with value 11 associated with the IP address of the Loopback0 interface.
- **SRGB**: NPF-XR1 signals the SRGB range, which is 16000-23999.
- **SR Algorithm**: This is a set of constraints that must be considered by the router (or SDN controller) upon SPF calculation. Initially, there were only two values: 0 (standard SPF process without any constraints) and 1 (SPF strict calculation). Today it is possible to add arbitrary algorithm values that correspond to constraints (for example, latency, hop count, QoS).

In a Cisco implementation, each router in an SR-MPLS domain signals the same set of information, so each router can calculate the label to reach the next SR router, which is calculated as SRGB Base + Prefix-SID. Example 16-3 provides details of the MPLS FIB from NPF-XR2.

Example 16-3 *Content of the MPLS FIB at NPF-XR2*

```
RP/0/0/CPU0:NPF-XR2#show mpls forwarding
```

```
Sun Mar 31 19:22:28.845 UTC

Local  Outgoing    Prefix             Outgoing     Next Hop        Bytes
Label  Label       or ID              Interface                    Switched
------ ----------- ------------------ ------------ --------------- ------------
16011  Pop         SR Pfx (idx 11)    Gi0/0/0/0    10.11.22.11     0
16033  Pop         SR Pfx (idx 33)    Gi0/0/0/1    10.22.33.33     0
24000  Pop         SR Adj (idx 1)     Gi0/0/0/1    10.22.33.33     0
24001  Pop         SR Adj (idx 3)     Gi0/0/0/1    10.22.33.33     0
24002  Pop         SR Adj (idx 1)     Gi0/0/0/0    10.11.22.11     0
24003  Pop         SR Adj (idx 3)     Gi0/0/0/0    10.11.22.11     0
```

Let's look at the MPLS label for NPF-XR1 as an example. The Local Label value 16011 is calculated as a sum of SRGB Base of the router NPF-XR2 itself (not NPF-XR1), which is 16000, and the Prefix-SID of IP address 10.0.0.11/32 learned from the IS-IS LSP of NPF-XR1, which is 11. Outgoing Label is calculated as the sum of SRGB Base from the adjacent router (NPF-XR1 in this case) and Prefix-SID of IP address 10.0.0.11/32 learned from the IS-IS LSP of NPF-XR1. You see the value Pop in this example because the egress router is directly connected, and NPF-XR2 performs a standard PHP operation.

You might wonder why each router takes a different SRGB Base value to compute the Local Label and Outgoing Label values. You need to understand what each label means. The local label is the value that the router (NPF-XR2 in this case) is looking for in incoming packets. The outgoing label value of NPF-XR2 is the local label value of NPF-XR1, which is the end destination for 10.0.0.11/32. Basically, each router calculates its own local and outgoing labels for all prefixes within the IGP domain. So, because the IS-IS database is consistent across the whole IS-IS domain, each router can calculate labels for each other router and implement traffic engineering.

There is no standardized value for SRGB in RFC 8402. This RFC recommends using the same value of SRGB across the whole SR-MPLS domain. There are a couple reasons for this, including consistency of operation and troubleshooting of the SR-MPLS operation. In reality, each vendor has a default SRGB, and to make Segment Routing interoperable between different vendors, the local calculation on each router must consider different SRGBs. To illustrate this scenario, Figure 16-5 shows the modified topology.

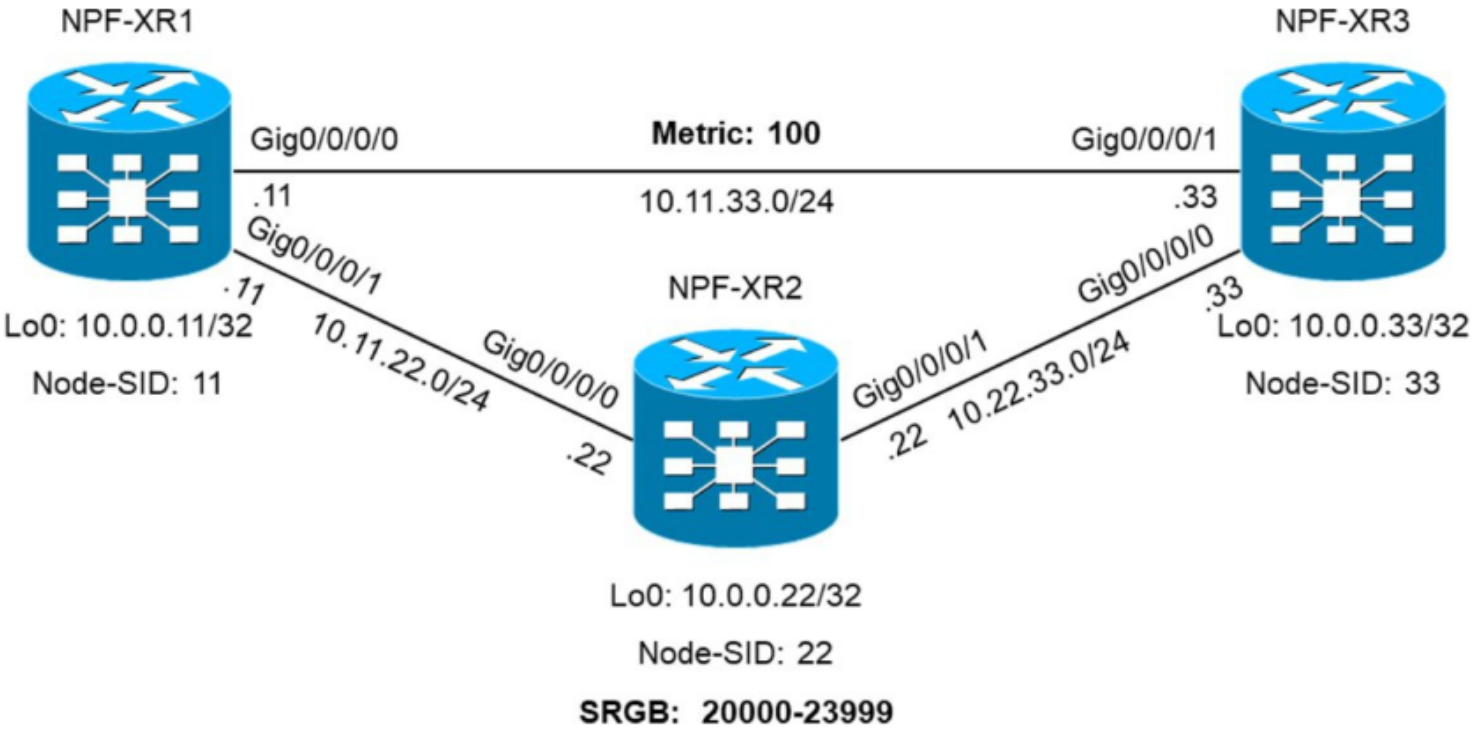

Figure 16-5 *Segment Routing Topology with a Modified Metric and SRGB*

The default IS-IS metric for all interfaces is 10, so putting 100 on the link between NPF-XR1 and NPF-XR3 makes all the traffic between these two nodes pass NPF-XR2. On NPF-XR2, the SRGB is altered to 20000-23999, as shown in Example 16-4.

Example 16-4 *Modifying the SRGB at NPF-XR2*

```
RP/0/0/CPU0:NPF-XR2(config)#show conf

Sun Mar 31 20:05:54.867 UTC

Building configuration...

!! IOS XR Configuration 6.5.1

router isis CORE

 segment-routing global-block 20000 23999

!

end
```

When the change is implemented, you can see in the MPLS FIB of NPF-XR2 that Local Label values are modified as shown in Example 16-5.

Example 16-5 *Updated MPLS FIB of NPF-XR2*

```
RP/0/0/CPU0:NPF-XR2#show mpls forwarding

Sun Mar 31 20:06:26.145 UTC
```

```
Local  Outgoing    Prefix             Outgoing     Next Hop        Bytes
Label  Label       or ID              Interface                    Switched
------ ----------- ------------------ ------------ --------------- ------------
20011  Pop         SR Pfx (idx 11)    Gi0/0/0/0    10.11.22.11     0
20033  Pop         SR Pfx (idx 33)    Gi0/0/0/1    10.22.33.33     0
24000  Pop         SR Adj (idx 1)     Gi0/0/0/1    10.22.33.33     0
24001  Pop         SR Adj (idx 3)     Gi0/0/0/1    10.22.33.33     0
24002  Pop         SR Adj (idx 1)     Gi0/0/0/0    10.11.22.11     0
24003  Pop         SR Adj (idx 3)     Gi0/0/0/0    10.11.22.11     0
```

The final step in this verification is to check the routing table and FIB on NPF-XR3 because the traffic to NPF-XR1 should flow over NPF-XR2. Example 16-6 shows several changes.

Example 16-6 *RIB and FIB on NPF-XR3 After Changes in the Network*

```
Routing entry for 10.0.0.11/32

  Known via "isis CORE", distance 115, metric 20, labeled SR, type level-2

  Installed Mar 31 20:08:01.783 for 00:03:10

  Routing Descriptor Blocks

    10.22.33.22, from 10.0.0.11, via GigabitEthernet0/0/0/0

      Route metric is 20

  No advertising protos.
```

```
RP/0/0/CPU0:NPF-XR3#show mpls forwarding

Sun Mar 31 20:11:21.150 UTC

Local  Outgoing    Prefix             Outgoing     Next Hop        Bytes
Label  Label       or ID              Interface                    Switched
------ ----------- ------------------ ------------ --------------- ------------
16011  20011       SR Pfx (idx 11)    Gi0/0/0/0    10.22.33.22     0
16022  Pop         SR Pfx (idx 22)    Gi0/0/0/0    10.22.33.22     0
24000  Pop         SR Adj (idx 1)     Gi0/0/0/0    10.22.33.22     0
24001  Pop         SR Adj (idx 3)     Gi0/0/0/0    10.22.33.22     0
24002  Pop         SR Adj (idx 1)     Gi0/0/0/1    10.11.33.11     0
24003  Pop         SR Adj (idx 3)     Gi0/0/0/1    10.11.33.11     0
```

As you can see, the next hop to 10.0.0.11/32 is NPF-XR2 now, and this is expected based on standard SPF calculation. Moreover, in the FIB, you can see that the Local Label value for NPF-XR3 is 16011 because the SRGB base wasn't changed locally, and the Outgoing Label value is 20011, which reflects the change of SRGB base on NPF-XR2. Example 16-7 shows how to use **traceroute** to check how the data plane looks when using MPLS.

Example 16-7 *Segment Routing Data Plane Verification*

```
RP/0/0/CPU0:NPF-XR3#traceroute mpls ipv4 10.0.0.11/32 source 10.0.0.33

Sun Mar 31 20:21:26.188 UTC


Tracing MPLS Label Switched Path to 10.0.0.11/32, timeout is 2 seconds


Codes: '!' - success, 'Q' - request not sent, '.' - timeout,

  'L' - labeled output interface, 'B' - unlabeled output interface,

  'D' - DS Map mismatch, 'F' - no FEC mapping, 'f' - FEC mismatch,

  'M' - malformed request, 'm' - unsupported tlvs, 'N' - no rx label,

  'P' - no rx intf label prot, 'p' - premature termination of LSP,

  'R' - transit router, 'I' - unknown upstream index,

  'X' - unknown return code, 'x' - return code 0


Type escape sequence to abort.


  0 10.22.33.33 MRU 1500 [Labels: 20011 Exp: 0]

L 1 10.22.33.22 MRU 1500 [Labels: implicit-null Exp: 0] 10 ms
```

```
! 2 10.11.22.11 10 ms
```

The outgoing MPLS label on NPF-XR3 is 20011, and NPF-XR2 performs PHP further, which is why you see implicit-null. If there were no PHP, the value would be 16011.

If the SRGB at NPF-XR2 is restored to the default value, then NPF-XR3 will have the same Local Label value as NPF-XR2, as shown in the Example 16-8. (This assumes that the changes from the Example 16-4 are reverted.)

Example 16-8 *SRGB Restored at NPF-XR2*

```
RP/0/0/CPU0:NPF-XR2#rollback configuration last 1

Sun Mar 31 21:06:49.907 UTC


Loading Rollback Changes.

Loaded Rollback Changes in 1 sec

Committing.RP/0/0/CPU0:Mar 31 21:06:51.636 UTC: isis[1011]: %ROUTING-ISIS-6-SRGB_INFO :

SRGB info: 'Segment routing temporarily disabled on all topologies and address families

because the global block is being modified'


4 items committed in 1 sec (4)items/sec

Updating.RP/0/0/CPU0:Mar 31 21:06:52.966 UTC: config_rollback[65725]: %MGBL-CONFIG-6-

DB_COMMIT : Configuration committed by user 'cisco'. Use 'show configuration commit

changes 1000000010' to view the changes.


Updated Commit database in 1 sec

Configuration successfully rolled back 1 commits.


RP/0/0/CPU0:NPF-XR3#show mpls forwarding

Sun Mar 31 21:08:37.524 UTC

Local  Outgoing    Prefix             Outgoing     Next Hop        Bytes

Label  Label       or ID              Interface                    Switched

------ ----------- ------------------ ------------ --------------- ------------

16011  16011       SR Pfx (idx 11)    Gi0/0/0/0    10.22.33.22     0

16022  Pop         SR Pfx (idx 22)    Gi0/0/0/0    10.22.33.22     0

24000  Pop         SR Adj (idx 1)     Gi0/0/0/0    10.22.33.22     0

24001  Pop         SR Adj (idx 3)     Gi0/0/0/0    10.22.33.22     0

24002  Pop         SR Adj (idx 1)     Gi0/0/0/1    10.11.33.11     0

24003  Pop         SR Adj (idx 3)     Gi0/0/0/1    10.11.33.11     0
```

The examples so far in this chapter demonstrate the standard operation of Segment Routing. A common misconception is that the ingress router uses the SR label signaled by the egress router through the whole SR-MPLS domain. In reality, each router calculates the Local and Outgoing labels itself, and these values can be different. However, if you have a single vendor, and you don't alter the default configuration, you might see that all the routers within an SR-MPLS domain have precisely the same MPLS label for a particular prefix; this is the case because the computation rules explained earlier are the same for each router. Figure 16-6 provides a higher-level view of Segment Routing with fewer details.

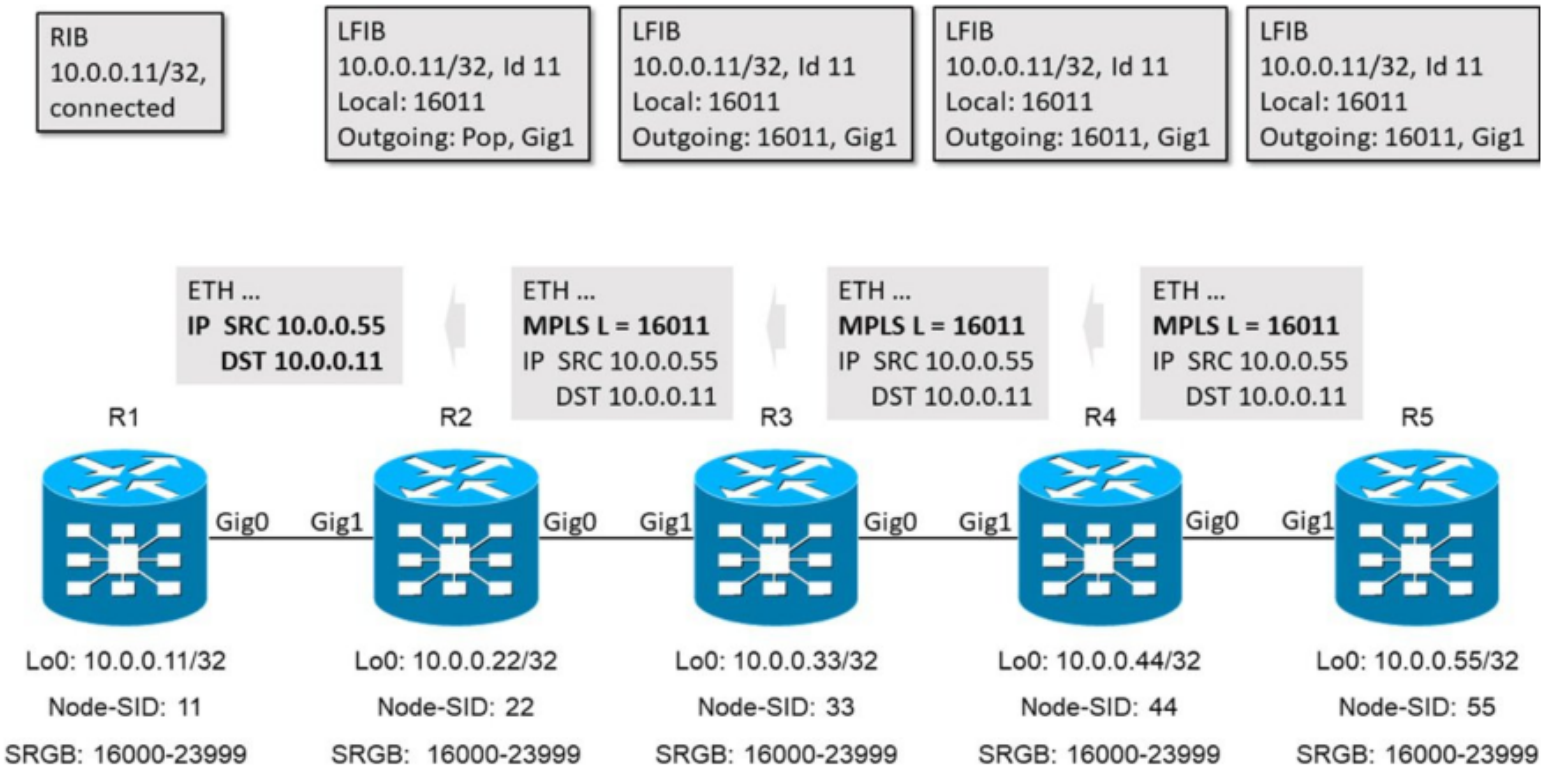

Figure 16-6 *Generic Process of Segment Routing Forwarding*

In this example, there is no additional signaling protocol in the network besides IS-IS (or OSPF, if you are running it) as all the MPLS information is encoded there. In traditional MPLS deployments, you have either LDP for non MPLS-TE case or RSVP in case of MPLS-TE in addition to your IGP. It's important to emphasize that, by default, Segment Routing follows standard SPF procedure, and ECMP is natively built in, which improves bandwidth utilization in the network.

The calculation consistency discussed here forms a solid basis for programming network paths using SR-TE, which is discussed in the next section.

Segment Routing Traffic Engineering

So far, the focus of this chapter has been on traffic forwarding without any constraints, which is suitable for enterprise locations, data centers, and traffic that isn't sensitive to latency. In the case of service providers offering converged services, including telephony or video conferencing, latency plays a crucial role. Therefore, traffic engineering technologies are an inevitable part of network design. Moreover, this is a perfect use case for Segment Routing because all the routers in an SR domain can calculate proper MPLS labels for any given prefix in the network by using Prefix-SID and SRGB Base. In addition, these routers know all the Adj-SID values (which are also MPLS labels), as they are signaled in IS-IS LSPs (as demonstrated in Example 16-2). Based on this information, there are two significant approaches to implementing traffic engineering using Segment Routing: via either Node-SID or Adj-SID. It is also possible to implement a third approach that mixes the first two. In the end, it's just a matter of ensuring a proper label stack on the head-end router, as this is precisely the way SR-TE works in the network. Figure 16-7 provides a high-level illustration of SR-TE on a network.

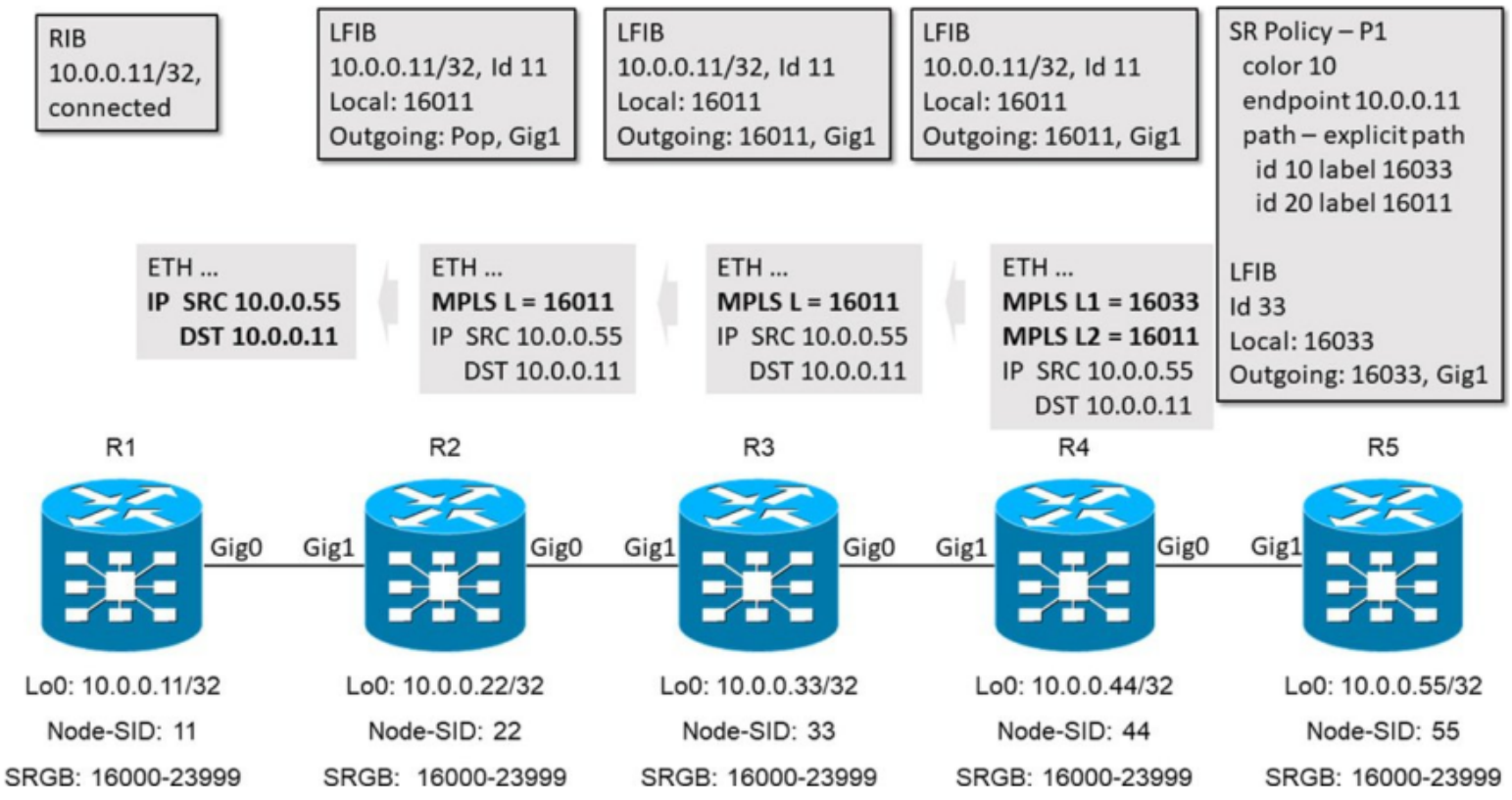

Figure 16-7 *Generic Process of Segment Routing Traffic Engineering Using Node-SID*

The critical component for SR-TE in Cisco IOS XR is SR-TE policies. Such a policy contains the following:

• **Name**: The name defines the configuration context.

• **Endpoint**: Within a policy, the endpoint defines the IPv4 address of next hop along the route. With BGP IP VPN construction (also known as IP/MPLS VPN or L3 VPN), the endpoint is equal to the IPv4 address of the loopback used as a next hop in the BGP update.

• **Color**: The color triggers the policy for a particular route. In a nutshell, it is an opaque extended community that can be added to any route announced via any BGP address family (for example, IPv4/IPv6 unicast, VPNv4/VPNv6) by using a route policy. Although BGP is mentioned as an example here, it is also possible to use static routes. The most significant advantage of this color extended community is that it provides a possibility to distinguish routes within the same IP VPN and to provide different paths over the network.

• **Path**: The final part of the SR-TE policy is the path, which defines the way the network routes the traffic. There are three available options: dynamic, constraints, and explicit. The explicit path equals to explicit-path value in MPLS-TE, where you define the specific set of next hops using either labels or IPv4 addresses. In the case of a dynamic path, it's the router itself; or, with the help of an SDN controller, it is possible to calculate the path using the defined metric type (either standard IGP or TE metric, or latency, and so on). The constraint is not the path itself; rather, the constraint specifies which additional parameters the router (or SDN controller) needs to take into account to compute the path. The algorithm for calculating the path under those constraints is called CSPF (Constrained Shortest Path First), and it is similar to classical SPF used in IS-IS/OSPF but with additional inputs rather than only a cost metric.

Although these details might seem complicated, the router configuration is quite straightforward. To illustrate it, we can use the reference topology from Figure 16-4 with a basic IS-IS and SR configuration and equal metrics on all the links. The SR-TE policy is configured using Node-SIDs for path definition as shown in Example 16-9.

Example 16-9 *SR-TE Policy Using Prefix-SIDs*

```
RP/0/0/CPU0:NPF-XR1#show run segment-routing

Wed Apr  3 05:58:32.600 UTC

segment-routing

 traffic-eng

  segment-list SRTE_XR1_XR3_SID

   index 10 mpls label 16022

   index 20 mpls label 16033

  !

  policy XR1_XR3_OVER_XR2_BLUE

   binding-sid mpls 1000002

   color 10 end-point ipv4 10.0.0.33

   candidate-paths

    preference 100

     explicit segment-list SRTE_XR1_XR3_SID

     !

    !

   !

  !

 !

!
```

Let's go step by step through the output shown in Example 16-9. First of all, you need to get into the configuration context for SR-TE policies by issuing the **segment-routing traffic-eng** command. You then have plenty of points to configure. Example 16-9 uses an explicit path definition, which is realized through the creation of a segment list. (The segment list is similar to an explicit path in an MPLS-TE configuration.) Inside the segment list, you need to provide either a sequence of MPLS labels or IPv4 next hops, which will be automatically converted into MPLS labels. When you complete the segment list, the next step is to create the policy itself. Inside the policy, you need to define of the following information:

• **binding-sid**: This uniquely codes the policy and represents it outside the router. Generally, it can work like any other local label, which means that if the router receives the packet labeled with this MPLS label, it replaces it with the label stack associated with the policy. In Example 16-9, the path through the network SRTE_XR1_XR3_SID is encoded using a label stack consisting of two labels: 16022 and 16033. Inside the policy XR1_XR3_OVER_XR2_blue, this path is mapped with the color and specific endpoint to the **binding-sid** value **1000002**. This binding-sid value is installed in the MPLS LFIB of the router; therefore, if NPF-XR1 receives from any neighbor the packet with label 1000002, it replaces it with two labels: 16022 and 16033. The core idea of this technology is to make the label stack that is used more shallow to cope with the hardware capability on the low-end routers.

• **color**: This, together with **end-point**, defines the set of prefixes that are tagged with this opaque extended community and the PE that these prefixes come from.

• **candidate-path**: This contains all the possible paths associated with the policy. It's possible to configure several paths with different **preference** values; if you do, the highest preference wins.

• **Path type**: The path type (dynamic, constraints, or explicit) is set within a particular path. For the explicit path type, the name of the segment list is defined; for the dynamic path type, the metric type is defined.

After you have configured a policy, you can check whether it is correct and working by using the verification command shown in Example 16-10.

Example 16-10 *Verifying the SR-TE Policy*

```
RP/0/0/CPU0:NPF-XR1#show segment-routing traffic-eng policy

Wed Apr  3 22:15:36.342 UTC


SR-TE policy database
```

```
--------------------

Name: XR1_XR3_OVER_XR2_BLUE (Color: 10, End-point: 10.0.0.33)
  Status:
    Admin: up  Operational: up for 01:20:10 (since Apr  3 20:55:26.152)
  Candidate-paths:
    Preference 100:
      Explicit: segment-list SRTE_XR1_XR3_SID (active)
        Weight: 1, Metric Type: IGP
          16022
          16033
  Attributes:
    Binding SID: 1000002
      Allocation mode: explicit
      State: Programmed
      Policy selected: yes
    Forward Class: 0
    Steering BGP disabled: no
    IPv6 caps enable: no
```

You can see in Example 16-10 that the policy is administratively and operationally up, but there are still no routes associated with the color 10. You can see how this policy works only if there are proper prefixes. In this case, you need to have BGP running, and you can extend the initial lab setup with BGP and route policy configuration as shown on Figure 16-8.

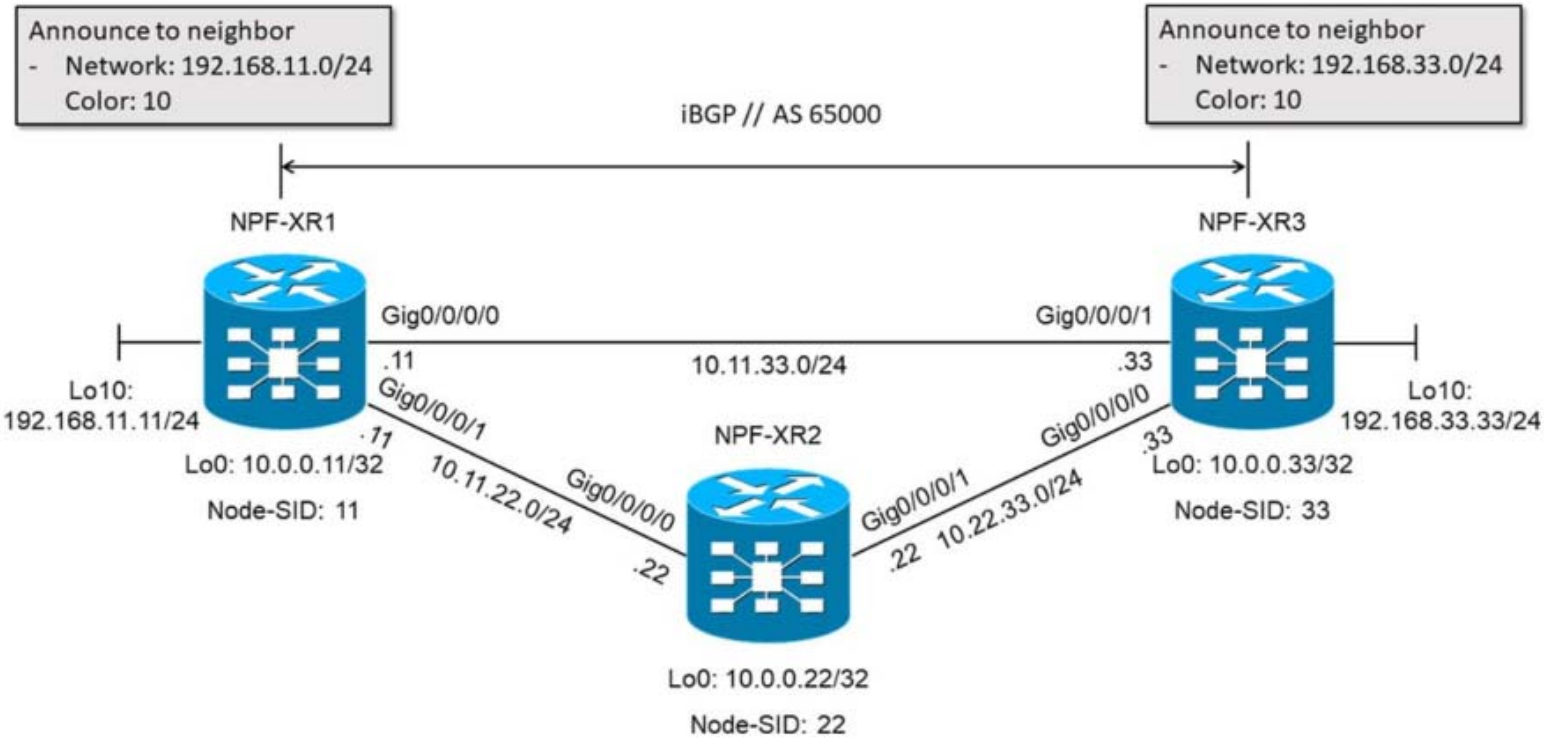

Figure 16-8 *BGP Configuration for a Color Extended Community*

The internal BGP (iBGP) peering is established between the Loopback0 interfaces of NPF-XR1 and NPF-XR3 routes within autonomous system 65000. The Loopback10 interfaces emulate customer prefixes that are advertised over BGP within the IPv4 unicast address family. These routes are tagged using the color opaque extended community. Example 16-11 shows the configuration of NPF-XR3.

Example 16-11 *BGP and Route Policy Configuration for a Color Extended Community*

```
RP/0/0/CPU0:NPF-XR3(config)#show conf
Wed Apr  3 05:58:32.600 UTC
interface Loopback10
 ipv4 address 192.168.33.33 255.255.255.0
!
extcommunity-set opaque COLOUR_BLUE
  10
end-set
!
```

```
route-policy RP_SET_COLOUR
  set extcommunity color COLOUR_BLUE
end-policy
!
router bgp 65000
 bgp router-id 10.0.0.33
 bgp log neighbor changes detail
 address-family ipv4 unicast
  network 192.168.33.0/24
 !
 neighbor 10.0.0.11
  remote-as 65000
  update-source Loopback0
  address-family ipv4 unicast
   route-policy RP_SET_COLOUR out
  !
 !
!
end
```

The configuration for NPF-XR1 isn't provided here because it is precisely the same except that it uses IPv4 addresses. Notice that the route policy operates with the community name; this is why you need to create **extcommunity-set opaque** *COMM_NAME* with the numeric value you plan to use in the SR-TE policy. Then you can create the proper route policy to call this set. The last step is to create a BGP process, announce the appropriate prefixes, and attach the route policy for the BGP neighbor within the corresponding address policy in the outgoing direction (though the incoming direction is also possible).

The SR-TE policy so far has been configured only on NPF-XR1 toward NPF-XR3, which is why the verification should be done on NFP-XR1. Example 16-12 shows the details of this verification.

Example 16-12 *Verifying the BGP Prefix Mapping to the SR-TE Policy*

```
RP/0/0/CPU0:NPF-XR1#show bgp ipv4 unicast 192.168.33.0/24
Wed Apr  3 22:22:33.514 UTC
BGP routing table entry for 192.168.33.0/24
Versions:
  Process           bRIB/RIB  SendTblVer
  Speaker                 10          10
Last Modified: Apr  3 20:55:29.543 for 01:27:03
Paths: (1 available, best #1)
  Not advertised to any peer
  Path #1: Received by speaker 0
  Not advertised to any peer
  Local
    10.0.0.33 C:10 (bsid:1000002) (metric 10) from 10.0.0.33 (10.0.0.33)
      Origin IGP, metric 0, localpref 100, valid, internal, best, group-best
      Received Path ID 0, Local Path ID 1, version 9
      Extended community: Color:10
      SR policy color 10, up, un-registered, bsid 1000002



RP/0/0/CPU0:NPF-XR1#show cef 192.168.33.00/24 detail
Wed Apr  3 22:22:35.944 UTC
192.168.33.0/24, version 21, internal 0x5000001 0x0 (ptr 0xa1416314) [1], 0x0 (0x0), 0x0 (0x0)
 Updated Apr  3 20:55:29.212
 Prefix Len 24, traffic index 0, precedence n/a, priority 4
  gateway array (0xa13357dc) reference count 1, flags 0x2010, source rib (7), 0 backups
                [1 type 3 flags 0x48441 (0xa13a0898) ext 0x0 (0x0)]
  LW-LDI[type=0, refc=0, ptr=0x0, sh-ldi=0x0]
  gateway array update type-time 1 Apr  3 20:55:29.212
```

```
LDI Update time Apr  3 20:55:29.212

 Level 1 - Load distribution: 0
 [0] via 10.0.0.33/32, recursive

  via local-label 1000002, 3 dependencies, recursive [flags 0x6000]
   path-idx 0 NHID 0x0 [0xa17cbc84 0x0]
   recursion-via-label
   next hop via 1000002/1/21

   Load distribution: 0 (refcount 1)

   Hash  OK  Interface                 Address
   0     Y   XR1_XR3_OVER_XR2_BLUE     point2point
```

The first command in Example 16-12 displays the information associated with the route. From an SR-TE policy perspective, the most crucial part is **extended community: color:10**. The router analyzes this community and provides a mapping to the locally configured SR **color** and **binding-SID** values associated with the SR-TE policy. If you compare the value shown here, 1000002, with the one configured in Example 16-9, you see that these values are identical.

The second command in Example 16-12 shows the details of the CEF, including the binding-SID value again together with the name of the SR-TE policy that is in use.

The SR-TE policy for the traffic routed from NPF-XR3 to NPF-XR1 can be configured by using Adj-SIDs, as mentioned earlier. Figure 16-9 illustrates the generic process of using the SR-TE policy with Adj-SIDs.

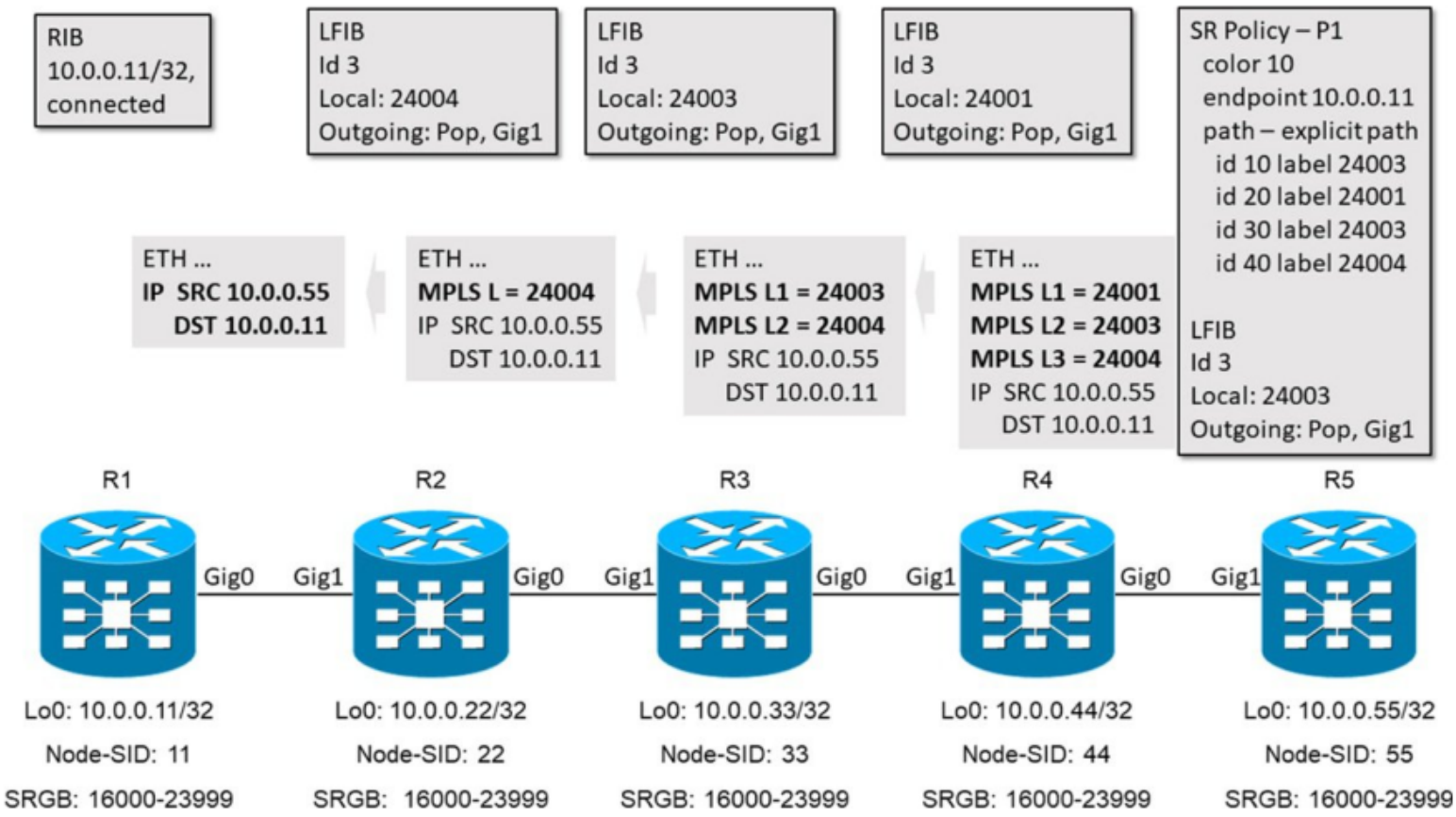

Figure 16-9 *Generic SR-TE Process Using Adj-SIDs*

As mentioned earlier, every network function in an SR-MPLS domain allocates an Adj-SID and, therefore, an MPLS label to each of its IGP neighbors. In Figure 16-9, the ID value from the LFB table is generated dynamically and has local significance just within the node. These values are also present in Example 16-8, which provides LFIB output. You might also have spotted that the Adj-SID labels come from the standard MPLS label range, which is also where LDP, RSVP-TE, and BGP labels come from. Because the outgoing label for the Adj-SID is always Pop, the SR-TE policy must contain Adj-SIDs for all hops on the path from the head-end router to the tail-end router.

A significant advantage of the approach just described is that the resulting path is exact, without any gray areas. For instance, if you use Prefix-SID, the local label is always the same within the node, regardless of the outgoing interface; in the case of tweaked per-link metrics, this might result in a different path than expected. Such a scenario with Adj-SID is just impossible. On the other hand, the major drawback of the SR-TE policies with Adj-SIDs is that there is a large label stack associated with the path. (The label stack contains all the labels attached to the packet for transmission.) In the Figure 16-9, you can see that four labels are imposed on the packet to traverse just four links. Considering that you would typically also have at least one service label (for an IP VPN or EVPN) and, possibly, an entropy label as well, the label stack can cross the hardware capabilities in terms of the label stack depth of many mid- to low-end routers. This is why you need to check what your hardware can do and decide which approach to use to reach your goal in terms of traffic engineering. For example, the label stack can be as small as 3 labels for a legacy platform and as high as 20 labels for a modern high-end router.

To solve the problem of large label stacks, the concept of MSD (Maximum SID Depth), which is described in RFC 8491, was introduced. Using this functionality, each platform participating in an SR domain advertises its MSD, which is a maximum possible allowed number of labels in a

stack that the network function can process in the routing protocol used inside the SR domain (for example, OSPF or IS-IS). Despite the different SR label stack creation approach, the configuration is absolutely the same, as you can these in Example 16-13.

Example 16-13 *SR-TE Policy Using Adj-SIDs*

```
RP/0/0/CPU0:NPF-XR3#show running-config segment-routing

Thu Apr  4 20:54:46.034 UTC

segment-routing

 traffic-eng

  segment-list SRTE_XR3_XR1_ADJ

   index 10 mpls label 24003

   index 20 mpls label 24001

  !

  policy XR3_XR1_OVER_XR2_BLUE

   binding-sid mpls 1000001

   color 10 end-point ipv4 10.0.0.11

   candidate-paths

    preference 100

     explicit segment-list SRTE_XR3_XR1_ADJ

     !

    !

   !

  !

 !

!
```

The configuration in Example 16-13 is pretty much the same as the one in Example 16-9. The only difference is the way the label stack is configured, as mentioned earlier.

To understand why these labels are chosen, take a look at Example 16-14, which provides LFIB content from NPF-XR3 and NPF-XR2.

Example 16-14 *Content of the LFIB from the Head End and the Transit Routers*

```
RP/0/0/CPU0:NPF-XR3#show mpls forwarding

Thu Apr  4 21:04:51.153 UTC

Local  Outgoing    Prefix             Outgoing     Next Hop        Bytes

Label  Label       or ID              Interface                    Switched

------ ----------- ------------------ ------------ --------------- ------------

16011  Pop         SR Pfx (idx 11)    Gi0/0/0/1    10.11.33.11     1244

16022  Pop         SR Pfx (idx 22)    Gi0/0/0/0    10.22.33.22     0

24000  Pop         SR Adj (idx 1)     Gi0/0/0/1    10.11.33.11     0

24001  Pop         SR Adj (idx 3)     Gi0/0/0/1    10.11.33.11     0

24002  Pop         SR Adj (idx 1)     Gi0/0/0/0    10.22.33.22     0

24003  Pop         SR Adj (idx 3)     Gi0/0/0/0    10.22.33.22     0

1000001 Pop         No ID              XR3_XR1_OVER point2point     0



RP/0/0/CPU0:NPF-XR2#show mpls forwarding

Thu Apr  4 21:05:20.629 UTC

Local  Outgoing    Prefix             Outgoing     Next Hop        Bytes

Label  Label       or ID              Interface                    Switched

------ ----------- ------------------ ------------ --------------- ------------

16011  Pop         SR Pfx (idx 11)    Gi0/0/0/0    10.11.22.11     0

16033  Pop         SR Pfx (idx 33)    Gi0/0/0/1    10.22.33.33     0

24000  Pop         SR Adj (idx 1)     Gi0/0/0/0    10.11.22.11     0

24001  Pop         SR Adj (idx 3)     Gi0/0/0/0    10.11.22.11     0

24002  Pop         SR Adj (idx 1)     Gi0/0/0/1    10.22.33.33     0

24003  Pop         SR Adj (idx 3)     Gi0/0/0/1    10.22.33.33     0
```

The Adj-SID labels used as the SR label stack for the corresponding SR-TE policy are shaded in Example 16-14. You might wonder why these labels are marked as idx 3 rather than idx 1. The answer lies in the IS-IS database, as shown in Example 16-15.

Example 16-15 *Adj-SIDs in the IS-IS Database*

```
RP/0/0/CPU0:NPF-XR2#show isis database verbose NPF-XR2.00-00

Thu Apr  4 21:37:24.118 UTC


IS-IS CORE (Level-2) Link State Database

LSPID                 LSP Seq Num  LSP Checksum  LSP Holdtime/Rcvd  ATT/P/OL

NPF-XR2.00-00       * 0x00000009   0x7373        657  /*            0/0/0

  Area Address:   49.0000

  Metric: 10         IS-Extended NPF-XR1.00

    Interface IP Address: 10.11.22.22

    Neighbor IP Address: 10.11.22.11

    Link Maximum SID Depth:

      Subtype: 1, Value: 10

    ADJ-SID: F:0 B:0 V:1 L:1 S:0 P:0 weight:0 Adjacency-sid:24001

  Metric: 10         IS-Extended NPF-XR3.00

    Interface IP Address: 10.22.33.22

    Neighbor IP Address: 10.22.33.33

    Link Maximum SID Depth:

      Subtype: 1, Value: 10

    ADJ-SID: F:0 B:0 V:1 L:1 S:0 P:0 weight:0 Adjacency-sid:24003

  NLPID:          0xcc

  IP Address:     10.0.0.22

  Metric: 0          IP-Extended 10.0.0.22/32

    Prefix-SID Index: 22, Algorithm:0, R:0 N:1 P:0 E:0 V:0 L:0

    Prefix Attribute Flags: X:0 R:0 N:1

  Hostname:       NPF-XR2

  Router Cap:     10.0.0.22, D:0, S:0

    Segment Routing: I:1 V:0, SRGB Base: 16000 Range: 8000

    SR Algorithm:

      Algorithm: 0

      Algorithm: 1

    Node Maximum SID Depth:

      Subtype: 1, Value: 10
```

By now, you should be familiar with both types of SR-TE policy configuration. In Example 16-12, you saw the commands you can use to verify whether an SR-TE policy is used for specific routes. Therefore, verification of the output is omitted in this example. The last step is to verify the data plane operation, as shown in Example 16-16.

Example 16-16 *SR-TE Data Plane Verification*

```
RP/0/0/CPU0:NPF-XR1#traceroute 192.168.33.33 source 192.168.11.11

Thu Apr  4 21:53:22.381 UTC


Type escape sequence to abort.

Tracing the route to 192.168.33.33


 1  10.11.22.22 [MPLS: Label 16033 Exp 0] 9 msec  0 msec  0 msec

 2  10.22.33.33 0 msec  *  0 msec



RP/0/0/CPU0:NPF-XR3#traceroute 192.168.11.11 source 192.168.33.33

Thu Apr  4 21:53:35.272 UTC


Type escape sequence to abort.

Tracing the route to 192.168.11.11


 1  10.22.33.22 [MPLS: Label 24001 Exp 0] 9 msec  0 msec  0 msec
```

```
2  10.11.22.11 0 msec  *  9 msec
```

In Example 16-16, you can see that in both SR-TE policies, the traffic passes through NPF-XR2, as planned, and this shows that the SR-TE policies are working correctly. You can also see the labels from the corresponding label stacks (Prefix-SID based and Adj-SID based), so you know that the segment lists are also working as expected.

You might wonder how everything in this section applies to you. Today, networking is moving from speaking of routers/switches to focusing on containers/services, and it's essential to be able to create a path for an application from the user to the content with an individual KPI. SR-TE makes this possible by allowing you to program the proper label stack on head-end routers or even on the end hosts within a data center. So far in this chapter, you have seen only manual configuration. Next, you will learn how to use programmable methods to collect Segment Routing information from a network and export it to SDN controllers and how to dynamically provision head-end routers in a programmatic way. Again, this programmability is built on a router's capability to program a path over the network by using SR-TE.

BGP Link State (BGP-LS)

The next step in the discovery of the programmability framework for service provider networks is to understand how the network states, including topology information and latency, can be collected. There are several ways to achieve this. The easiest way is to integrate an SDN controller in the IGP domain so that it can directly listen for IGP updates. The main drawback of this method is that the SDN controller will have visibility into only a single domain and won't be able to see the topology from other domains. The best approach today is to use a specific address family of BGP called BGP-LS.

BGP-LS Basics

BGP-LS is a new address family (or multiprotocol extension) for BGP. (The AFI/SAFI indicator is 16388/71.) RFC 7752 defines BGP-LS, and you can refer to it if you want to learn about each bit and byte.

In a nutshell, BGP-LS allows you to convert an IGP topology running IS-IS or OSPFv2/v3 into BGP routes, officially called BGP NLRI (network layer reachability information), and transmit this information to the SDN controller. The SDN controller can be located anywhere, and there is no need to connect it to the IGP domain directly. Moreover, the whole IGP topology—that is, the whole IS-IS or OSPF link state database (LSDB)—is exported over a single session, although it's recommended to have at least two sessions for redundancy. Figure 16-10 shows a high-level overview of BGP-LS peering.

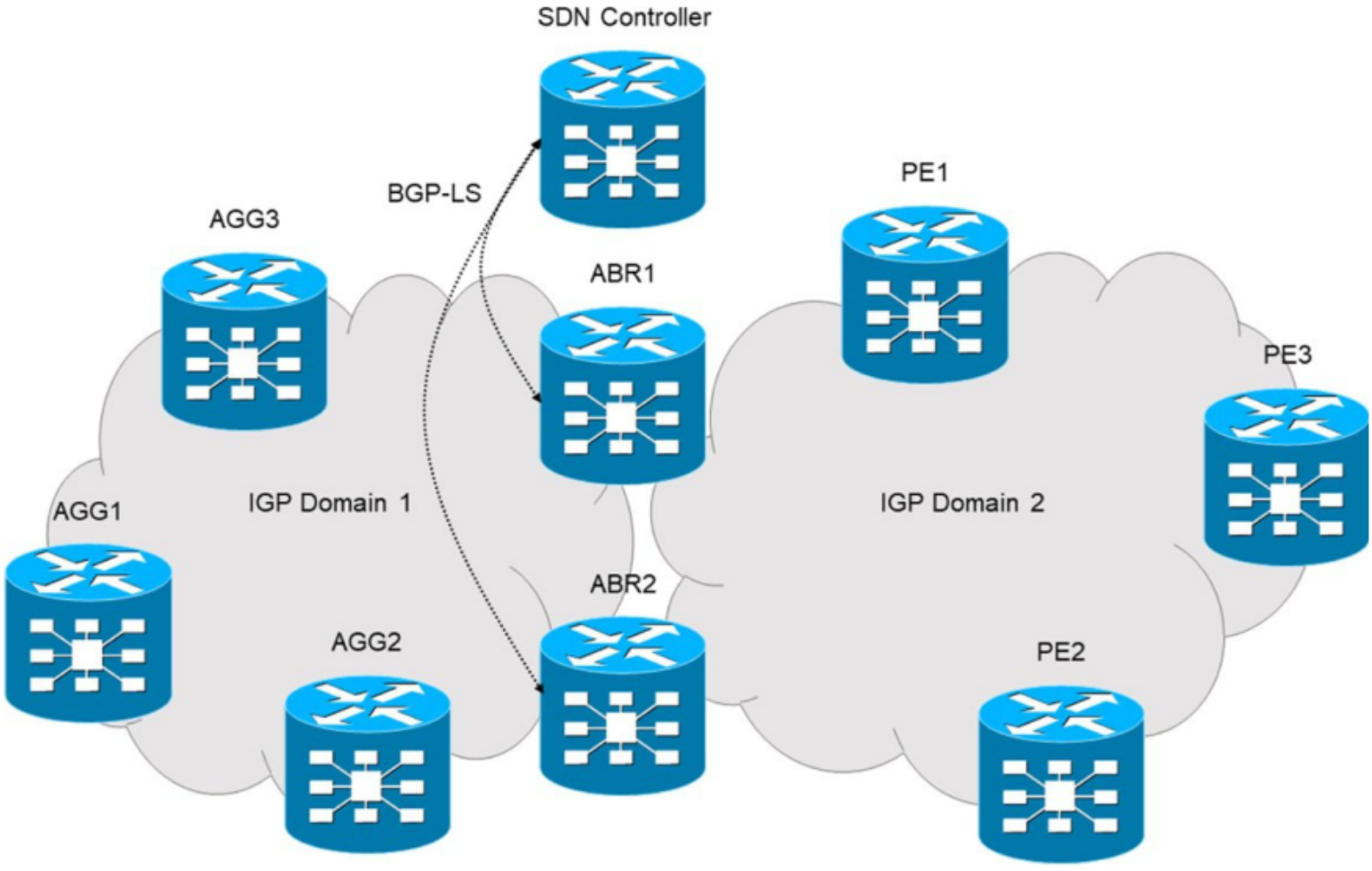

Figure 16-10 *High-Level BGP-LS Peering Architecture*

ABRs are typically the best choice for establishing BGP-LS peering with an SDN controller because ABRs have visibility in multiple IGP domains. These are the most common combinations:

- IGP Domain 1 is IS-IS process 0, level 1, and IGP Domain 2 is IS-IS process 0, level 2.
- IGP Domain 1 is IS-IS process 1, level 2, and IGP Domain 2 is IS-IS process 0, level 2.
- IGP Domain 1 is OSPF process 0, area nonzero (1, 2, and so on), and IGP Domain 2 is OSPF process 0, area 0.
- IGP Domain 1 is OSPF process 1, area 0, and IGP Domain 2 is OSPF process 0, area 0.

Other combinations are also possible.

A BGP session can be built using either internal or external design option. Which you choose doesn't matter too much, and you should follow the logic of your network:

- If you have a single AS for all your BGP domains, it makes sense to establish an iBGP session from the ABRs to your SDN controller.

• If you have multiple AS numbers (for example, if each IGP domain has its own AS), you can have the SDN controller in its own AS as well.

You should pay attention to standard BGP rules when building connectivity to an SDN controller (for example, a split-horizon rule for iBGP, a multihop rule for eBGP).

Before configuring your first BGP-LS session, it's essential to understand some security considerations. The message flow in terms of the BGP NLRI is unidirectional from the network functions in your network running IGP to the SDN controller. You should drop all incoming BGP-LS updates on your routers and filter all the outgoing BGP-LS updates on the SDN controller (if possible).

Figure 16-11 shows a simple lab topology used to configure BGP-LS in this section.

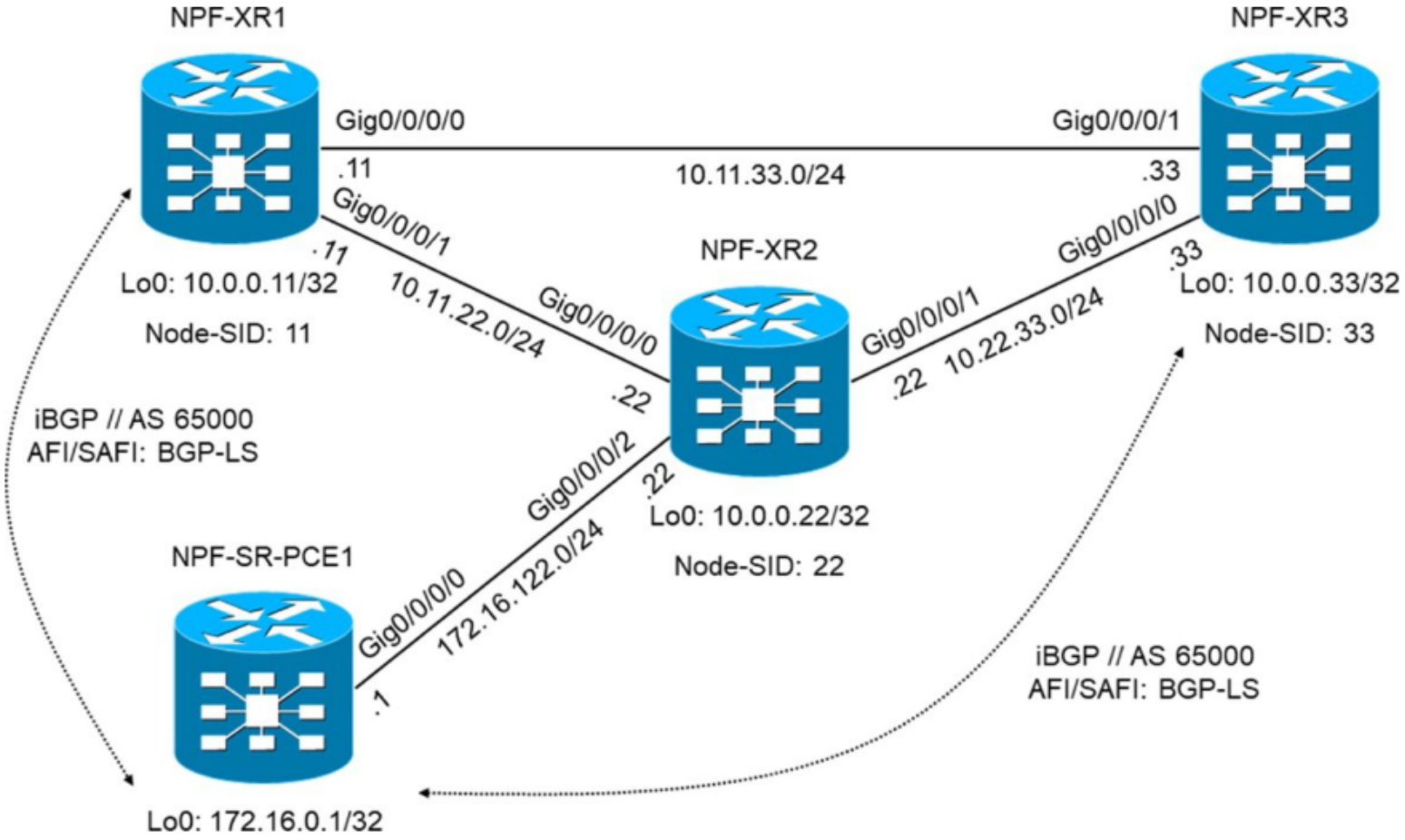

Figure 16-11 *Lab Topology for BGP-LS*

As you can see, this topology is the same as the topology used for Segment Routing earlier in this chapter (refer to Figure 16-4) —and this makes sense because the SDN controller should extend the capability of Segment Routing. The Cisco SDN controller, which calculates end-to-end MPLS paths, is called SR-PCE now; previously, it was known as XTC (XR Traffic Controller). In terms of connectivity, note the following configurations:

• The direct link between NPF-XR2 and NPF-SR-PCE1 isn't propagated into the core IS-IS process.

• NPF-XR2 has a static route pointing to NPF-SR-PCE1 Loopback0, which is propagated into the core IS-IS.

• NPF-SR-PCE1 has a default route pointing toward NPF-SR-PCE1 over a direct link.

• There are two internal BGP sessions, between NPF-SR-PCE1 and NPF-XR1 and between NPF-SR-PCE1 and NPF-XR2.

• The route filtering is applied such that NPF-XR1 and NPF-XR2 can only send BGP-LS updates, and NPF-SR-PCE1 can only receive them.

As you can see, this scenario is relatively easy from a configuration perspective. However, it requires two major code blocks: one to establish IP connectivity between network elements (that is, configure routing) and another to establish BGP-LS peering.

Example 16-17 shows the details for the first code block, which establishes IP connectivity between the network function from the lab and the SDN controller.

Example 16-17 *Configuring Basic Routing Between Routers and an SDN Controller*

```
RP/0/0/CPU0:NPF-XR2(config)#show conf

Sat Apr 20 12:48:02.226 UTC

Building configuration...

!! IOS XR Configuration 6.5.1

interface GigabitEthernet0/0/0/2

 ipv4 address 172.16.122.22 255.255.255.0

 no shutdown

!

!

prefix-set PS_SR_PCE_LO

  172.16.0.1/32
```

```
end-set
!
route-policy RP_STATIC_TO_ISIS
  if destination in PS_SR_PCE_LO then
    pass
  endif
end-policy
!
router static
 address-family ipv4 unicast
  172.16.0.1/32 GigabitEthernet0/0/0/2 172.16.122.1
 !
!
router isis CORE
 address-family ipv4 unicast
  redistribute static route-policy RP_STATIC_TO_ISIS
 !
!
end
```

```
RP/0/0/CPU0:NPF-SR-PCE1(config)#show conf
Sat Apr 20 13:02:08.597 UTC
Building configuration...
!! IOS XR Configuration 6.5.1
interface Loopback0
 ipv4 address 172.16.0.1 255.255.255.255
!
interface GigabitEthernet0/0/0/0
 ipv4 address 172.16.122.1 255.255.255.0
 no shutdown
!
router static
 address-family ipv4 unicast
  0.0.0.0/0 GigabitEthernet0/0/0/0 172.16.122.22
 !
!
end
```

Note

If you are familiar with the Cisco IOS XR operating system, Example 16-17 should make sense to you, as it doesn't touch on the BGP session yet but just shows the essential connectivity. If Example 16-17 does not make sense to you, you may want to spend some time trying to brush up on Cisco IOS XR and/or Cisco NX-OS.

Example 16-18 shows the configuration of the internal BGP session for the BGP-LS address family.

Example 16-18 *Configuring BGP-LS Peering*

```
RP/0/0/CPU0:NPF-XR1(config)#show conf
Sat Apr 20 17:27:14.291 UTC
Building configuration...
!! IOS XR Configuration 6.5.1
!
route-policy RP_DROP_ALL
  drop
end-policy
!
```

```
route-policy RP_PASS_ALL
  pass
end-policy
!
router bgp 65000
 bgp router-id 10.0.0.11
 bgp log neighbor changes detail
 address-family link-state link-state
 !
 neighbor 172.16.0.1
  update-source Loopback0
  remote-as 65000
  address-family link-state link-state
   route-policy RP_DROP_ALL in
   route-policy RP_PASS_ALL out
  !
 !
!
end



RP/0/0/CPU0:NPF-SR-PCE1(config)#show conf
Sat Apr 20 17:32:59.114 UTC
Building configuration...
!! IOS XR Configuration 6.5.1.34I
!
route-policy RP_DROP_ALL
  drop
end-policy
!
route-policy RP_PASS_ALL
  pass
end-policy
!
router bgp 65000
 bgp router-id 172.16.0.1
 bgp log neighbor changes detail
 address-family link-state link-state
 !
 neighbor 10.0.0.11
  remote-as 65000
  update-source Loopback0
  address-family link-state link-state
   route-policy RP_PASS_ALL in
   route-policy RP_DROP_ALL out
  !
 !
 neighbor 10.0.0.33
  remote-as 65000
  update-source Loopback0
  address-family link-state link-state
   route-policy RP_PASS_ALL in
   route-policy RP_DROP_ALL out
```

```
 !
!
!
end
```

Example 16-18 shows the configuration of a peering session between NPF-XR1 and NPF-SR-PCE1. (The configuration of NPF-XR3 is omitted from this section for the sake of brevity.) If you are familiar with BGP configuration on the Cisco IOS XR platform, the provided output should be quite easy to understand: You enable the proper address family within the BGP context and then enable peering with the corresponding neighbor. The configuration of the route policies isn't mandatory, but it helps to improve the stability, as explained previously. Example 16-19 shows the status of the established BGP sessions from the NPF-SR-PCE1 perspective.

Example 16-19 *Overview of BGP-LS Peering*

```
RP/0/0/CPU0:NPF-SR-PCE1#show bgp link-state link-state summary

Sat Apr 20 19:24:44.834 UTC

BGP router identifier 172.16.0.1, local AS number 65000

BGP generic scan interval 60 secs

Non-stop routing is enabled

BGP table state: Active

Table ID: 0x0   RD version: 0

BGP main routing table version 1

BGP NSR Initial initsync version 1 (Reached)

BGP NSR/ISSU Sync-Group versions 0/0

BGP scan interval 60 secs


BGP is operating in STANDALONE mode.



Process       RcvTblVer   bRIB/RIB   LabelVer  ImportVer  SendTblVer  StandbyVer

Speaker               1          1          1          1           1           0


Neighbor        Spk    AS MsgRcvd MsgSent   TblVer  InQ OutQ  Up/Down  St/PfxRcd

10.0.0.11         0 65000     111      111        1    0    0 01:48:50          0

10.0.0.33         0 65000     111      111        1    0    0 01:48:28          0
```

You can see in Example 16-19 that the status of the sessions is up, but there is no prefix received. This is correct because so far there has been no configuration that defines which IGP information should be exported to the SDN controller.

BGP-LS Route Types

Before we go further into the BGP-LS NLRI types, it's essential that you understand one more detail. Upon the export of IGP topology to the SDN controller, there must be some additional parameters that make it clear which IGP domain routes belong to. Because several IGPs might be configured on a single router, each IGP domain should have an identifier so that the SDN controller can match the routing information to a specific domain. This identifier is called an *instance identifier*, and it is part of the configuration of the export process.

Another important aspect is the idea of abstraction. IS-IS and OSPF have different structures for their link-state databases. However, an SDN controller should have an end-to-end view of the topology and should be able to program MPLS LSP regardless of the source IGP. This is why one of the core ideas of BGP-LS is to create BGP-LS routes that are as similar as possible for IS-IS and OSPF. This is achieved by using a standard BGP approach for information encoding using TLV (type length value) triples. A TLV defines a particular piece of information, where the type codes the meaning of the information, the length indicates how many subsequent octets should be analyzed, and the value is the parameter itself. TLV is a very flexible mechanism, as it makes it possible to create a single type values for certain information from both IS-IS and OSPF (for example, a router, link IP addresses) and to create additional TLV triples for IGP-specific information. All the BGP updates (not only BGP-LS) are built using the TLV approach.

There are officially four BGP-LS route types (although, technically, the third and fourth types listed here are equal but used for different address families):

- **Node NLRI**: This type represents the node in the IGP topology, which is similar to the node in the traffic engineering database (TED) with MPLS-TE.

- **Link NLRI**: This type represents the connectivity between the nodes defined in Node NLRI.

- **IPv4 topology prefix NLRI**: This type is used to encode an IPv4 address associated with any interface or redistributed from another protocol.

- **IPv6 topology prefix NLRI**: This type is used to encode an IPv6 address associated with any interface or redistributed from another protocol.

Each of these route types has some set of mandatory and optional TLV triples. Next, we will look into details of each route, using real examples. To have such routes, the routers NPF-XR1 and NPF-XR3 should export the IS-IS database into BGP-LS, as shown in Figure 16-11.

Example 16-20 shows the configuration that needs to be implemented for NPF-XR2 is shown. (Although it is not shown, the configuration for NPF-XR3 is similar.)

Example 16-20 *Distributing IS-IS to BGP-LS*

```
RP/0/0/CPU0:NPF-XR1(config-isis)#show conf

Sat Apr 20 20:34:22.082 UTC

Building configuration...

!! IOS XR Configuration 6.5.1

router isis CORE

 distribute link-state instance-id 32 level 2

!

end
```

This configuration is straightforward: Under the IGP that you want to redistribute over BGP-LS, you configure the corresponding command, including the instance identifier, and, optionally, provide some further details (for example, the IS-IS Level 2–only routes in Example 16-20). When the configuration is applied on both routers, the IS-IS routes are converted into BGP-LS updates and sent to the SDN controller. Using the verification command on NPF-SR-PCE1, you can see that routes are propagated (see Example 16-21).

Example 16-21 *Propagating the BGP-LS Routes*

```
RP/0/0/CPU0:NPF-SR-PCE1#show bgp link-state link-state summary

Sat Apr 20 20:44:56.475 UTC

BGP router identifier 172.16.0.1, local AS number 65000

BGP generic scan interval 60 secs

Non-stop routing is enabled

BGP table state: Active

Table ID: 0x0   RD version: 0

BGP main routing table version 1

BGP NSR Initial initsync version 1 (Reached)

BGP NSR/ISSU Sync-Group versions 0/0

BGP scan interval 60 secs


BGP is operating in STANDALONE mode.



Process       RcvTblVer   bRIB/RIB   LabelVer  ImportVer  SendTblVer  StandbyVer

Speaker               1          1          1          1           1           0


Neighbor        Spk    AS MsgRcvd MsgSent   TblVer  InQ OutQ  Up/Down  St/PfxRcd

10.0.0.11         0 65000     201     191        1    0    0 03:09:02         13

10.0.0.33         0 65000     201     191        1    0    0 03:08:40         13
```

As you can see in Example 16-21, NPF-XR1 and NPF-XR2 send 13 routes each. These 13 routes contain all three types of BGP-LS routes: node, link, and prefix. Example 16-22 shows some details of these routes.

Example 16-22 *Content of the BGP-LS RIB*

```
RP/0/0/CPU0:NPF-SR-PCE1#show bgp link-state link-state

Sat Apr 20 22:39:05.665 UTC

BGP router identifier 172.16.0.1, local AS number 65000

BGP generic scan interval 60 secs

Non-stop routing is enabled

BGP table state: Active

Table ID: 0x0   RD version: 0

BGP main routing table version 1

BGP NSR Initial initsync version 1 (Reached)

BGP NSR/ISSU Sync-Group versions 0/0

BGP scan interval 60 secs


Status codes: s suppressed, d damped, h history, * valid, > best

              i - internal, r RIB-failure, S stale, N Nexthop-discard

Origin codes: i - IGP, e - EGP, ? - incomplete

Prefix codes: E link, V node, T IP reacheable route, u/U unknown

              I Identifier, N local node, R remote node, L link, P prefix

              L1/L2 ISIS level-1/level-2, O OSPF, D direct, S static/peer-node
```

```
              a area-ID, l link-ID, t topology-ID, s ISO-ID,
              c confed-ID/ASN, b bgp-identifier, r router-ID,
              i if-address, n nbr-address, o OSPF Route-type, p IP-prefix
              d designated router address
   Network            Next Hop            Metric LocPrf Weight Path
* i[V][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0011.00]]/328
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[V][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0022.00]]/328
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[V][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0033.00]]/328
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[E][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][R[c65000][b0.0.0.0][s0100.0000
.0022.00]][L[i10.11.22.11][n10.11.22.22]]/696
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[E][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][R[c65000][b0.0.0.0][s0100.0000
.0033.00]][L[i10.11.33.11][n10.11.33.33]]/696
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[E][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0022.00]][R[c65000][b0.0.0.0][s0100.0000
.0011.00]][L[i10.11.22.22][n10.11.22.11]]/696
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[E][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0022.00]][R[c65000][b0.0.0.0][s0100.0000
.0033.00]][L[i10.22.33.22][n10.22.33.33]]/696
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[E][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0033.00]][R[c65000][b0.0.0.0][s0100.0000
.0011.00]][L[i10.11.33.33][n10.11.33.11]]/696
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[E][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0033.00]][R[c65000][b0.0.0.0][s0100.0000
.0022.00]][L[i10.22.33.33][n10.22.33.22]]/696
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[T][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][P[p10.0.0.11/32]]/400
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[T][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0022.00]][P[p10.0.0.22/32]]/400
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[T][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0022.00]][P[p172.16.0.1/32]]/400
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
* i[T][L2][IOx20][N[c65000][b0.0.0.0][s0100.0000.0033.00]][P[p10.0.0.33/32]]/400
                      10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
```

As mentioned earlier, Example 16-22 shows 13 routes, which are classified in the following manner:

• [V] routes are node NLRIs, so they represent nodes in the IGP topology. There are three nodes (NPF-XR1, NPF-XR2, and NPF-XR3); hence, there are three node NLRIs.

• [E] routes are link NLRIs, and they describe connectivity between nodes from [V] routes. Strictly speaking, each [E] documents a half-link, meaning the link parameters from the perspective of the particular node. This is why the full bidirectional link is described by a pair of [E] routes, exactly as is done in IS-IS or OSPF. There are three nodes connected in the topology shown in Figure 16-11; therefore, there are three bidirectional links or six unidirectional (half) links, which equals six [E] routes in the BGP-LS RIB.

• [T] routes are prefix NLRIs, which contain all IPv4 or IPv6 addresses (if present in the topology) advertised in the IGP. According to the lab configuration shown in Figure 16-11, IS-IS is running only for the IPv4 address family, and only Loopback0 IP addresses are advertised. In addition, there is one static route pointing to NPF-SR-PCE1 and distributed into IS-IS, so there are four IP addresses in the IGP topology and four [T] routes in the BGP-LS RIB.

Now that you have learned about route types in general, it's a good time to look at the details that can help you figure out what information is provided.

Node NLRI

Example 16-23 shows the details of the node NLRI.

Example 16-23 *Information in a Node NLRI*

```
RP/0/0/CPU0:NPF-SR-PCE1#show bgp link-state link-state

! The output is truncated for brevity

Status codes: s suppressed, d damped, h history, * valid, > best

              i - internal, r RIB-failure, S stale, N Nexthop-discard

Origin codes: i - IGP, e - EGP, ? - incomplete

Prefix codes: E link, V node, T IP reacheable route, u/U unknown

              I Identifier, N local node, R remote node, L link, P prefix

              L1/L2 ISIS level-1/level-2, O OSPF, D direct, S static/peer-node

              a area-ID, l link-ID, t topology-ID, s ISO-ID,

              c confed-ID/ASN, b bgp-identifier, r router-ID,

              i if-address, n nbr-address, o OSPF Route-type, p IP-prefix

              d designated router address

   Network            Next Hop            Metric LocPrf Weight Path

* i[V][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]]/328

                     10.0.0.11                100      0 i

* i                  10.0.0.33                100      0 i

! Further output is truncated for brevity



RP/0/0/CPU0:NPF-SR-PCE1#show bgp link-state link-state

[V][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]]/328

Sun Apr 21 10:03:24.983 UTC

BGP routing table entry for [V][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]]/328

Versions:

  Process           bRIB/RIB  SendTblVer

  Speaker                  0           0

Last Modified: Apr 20 20:34:30.541 for 13:28:54

Paths: (2 available, no best path)

  Not advertised to any peer

  Path #1: Received by speaker 0

  Not advertised to any peer

  Local

    10.0.0.11 (inaccessible) from 10.0.0.11 (10.0.0.11)

      Origin IGP, localpref 100, valid, internal

      Received Path ID 0, Local Path ID 0, version 0

      Link-state: Node-name: NPF-XR1, ISIS area: 49.00.00, SRGB: 16000:8000

                  SR-ALG: 0 SR-ALG: 1 , SRLB: 15000:1000 , MSD: Type 1 Value 10


  Path #2: Received by speaker 0

  Not advertised to any peer

  Local

    10.0.0.33 (inaccessible) from 10.0.0.33 (10.0.0.33)

      Origin IGP, localpref 100, valid, internal
```

```
      Received Path ID 0, Local Path ID 0, version 0

      Link-state: Node-name: NPF-XR1, ISIS area: 49.00.00, SRGB: 16000:8000

                  SR-ALG: 0 SR-ALG: 1 , SRLB: 15000:1000 , MSD: Type 1 Value 10
```

Before we look at the content of the route itself, we need to examine the route naming. The legend provided at the beginning of the full BGP-LS RIB output is very detailed and helps you easily read the route. For instance, from the route [V][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]]/328, you can glean the following information:

- **V**: This indicates a node NLRI.
- **L2**: This indicates IS-IS Level 2.
- **I0x20**: The instance-id is 32 when configured during an export on NPF-XR1 and NPF-XR3. (0x20 in hexadecimal equals 32 in decimal.)
- **N**: This describes the local node. (There is also a remote node, which is covered later in this section.)
- **c65000**: This is the BGP ASN of the router exporting information about this node.
- **b0.0.0.0**: This is the BGP-LS identifier (which is all zeros by default); it can be altered in complex topologies.
- **s0100.0000.0011.00**: This is the node part of IS-IS NET without the area part.
- **328**: This is the overall length of the route, in bytes. It is really a cosmetic entry, as it is not meaningful from a user standpoint and is no longer visible in newer Cisco IOS XR releases. (In further route output in this chapter, this parameter is omitted from explanation.)

Even just this information gives you some insights into the topology. In addition, the second part of Example 16-23 provides further facts about the node, including the hostname and important Segment Routing parameters, such as Segment Routing Global/Local Block, supported SR algorithms, and MSD (which is type 1 with value 10). If you compare this information with the content of the IS-IS LSDB in Example 16-2, you can see that all the fields not directly related to IP addresses (IP reachability information) or links (IS-Extended) are copied to this node NLRI.

Link NLRI

Example 16-24 shows the information provided in a link NLRI, which connects node NLRIs.

Example 16-24 *Information in a Link NLRI*

```
RP/0/0/CPU0:NPF-SR-PCE1#show bgp link-state link-state

! The output is truncated for brevity

Status codes: s suppressed, d damped, h history, * valid, > best

              i - internal, r RIB-failure, S stale, N Nexthop-discard

Origin codes: i - IGP, e - EGP, ? - incomplete

Prefix codes: E link, V node, T IP reacheable route, u/U unknown

              I Identifier, N local node, R remote node, L link, P prefix

              L1/L2 ISIS level-1/level-2, O OSPF, D direct, S static/peer-node

              a area-ID, l link-ID, t topology-ID, s ISO-ID,

              c confed-ID/ASN, b bgp-identifier, r router-ID,

              i if-address, n nbr-address, o OSPF Route-type, p IP-prefix

              d designated router address

   Network            Next Hop            Metric LocPrf Weight Path

* i[E][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][R[c65000][b0.0.0.0][s0100.0000

.0022.00]][L[i10.11.22.11][n10.11.22.22]]/696

                     10.0.0.11                     100      0 i

* i                  10.0.0.33                     100      0 i

! Further output is truncated for brevity




RP/0/0/CPU0:NPF-SR-PCE1# show bgp link-state link-state

[E][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][R[c65000][b0.0.0.0][s0100.0000.

0022.00]][L[i10.11.22.11][n10.11.22.22]]/696

Sun Apr 21 10:31:47.446 UTC

BGP routing table entry for

[E][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][R[c65000][b0.0.0.0][s0100.0000.

0022.00]][L[i10.11.22.11][n10.11.22.22]]/696

Versions:

  Process           bRIB/RIB  SendTblVer

  Speaker                  0           0

Last Modified: Apr 20 20:34:30.541 for 13:57:16

Paths: (2 available, no best path)
```

```
  Not advertised to any peer

  Path #1: Received by speaker 0

  Not advertised to any peer

  Local

    10.0.0.11 (inaccessible) from 10.0.0.11 (10.0.0.11)

      Origin IGP, localpref 100, valid, internal

      Received Path ID 0, Local Path ID 0, version 0

      Link-state: metric: 10, ADJ-SID: 24003(30) , MSD: Type 1 Value 10


  Path #2: Received by speaker 0

  Not advertised to any peer

  Local

    10.0.0.33 (inaccessible) from 10.0.0.33 (10.0.0.33)

      Origin IGP, localpref 100, valid, internal

      Received Path ID 0, Local Path ID 0, version 0

      Link-state: metric: 10, ADJ-SID: 24003(30) , MSD: Type 1 Value 10
```

Following the same approach as for the node NLRI, for the link NLRI, we start with the route itself. From the link NLRI [E][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][R[c65000][b0.0.0.0][s0100.0000.0022.00]][L[i10.11.22.11][n10.11.22.22]]/696, you can glean the following information:

- **E**: This indicates a link NLRI.
- **L2**: This indicates IS-IS Level 2.
- **I0x20**: The instance-id is 32 when configured during an export on NPF-XR1 and NPF-XR3. (0x20 in hexadecimal equals 32 in decimal.)
- **N**: This describes the local node, and connectivity is described from the perspective of this node. (That is, the information is extracted from IS-IS LSP for this node.)
- **c65000**: This is the BGP ASN of the router exporting information about this node.
- **b0.0.0.0**: This is the BGP-LS identifier (which is all zeros by default), and it can be altered in complex topologies.
- **s0100.0000.0011.00**: This is the node part of IS-IS NET, without the area part.
- **R**: This is the remote node, or the link endpoint to which the local node is connected.
- **L**: This indicates that parameters of the link itself are provided.
- **i10.11.22.11**: This is the interface IP address of the local node.
- **n10.11.22.22**: This is the neighbor IP address, which is basically the interface IP address of the remote node.
- **696**: This is the overall length of the route, in bytes.

By this time, you should be starting to see that by looking at BGP-LS routes, you can construct the topology just as the SDN controller does. In the second part of the output in Example 16-24, you see further details, including metrics associated with the link, Segment Routing Adj-SID (MPLS label), and MSD. Generally, the content of the link NLRI is extracted from the IS-Extended entries of the IS-IS LSP, as shown in the Example 16-2.

Prefix NLRI

The third BGP-LS route type is the prefix NLRI, which contains the IP reachability information and related parameters (see Example 16-25).

Example 16-25 *Information in a Prefix NLRI*

```
RP/0/0/CPU0:NPF-SR-PCE1#show bgp link-state link-state

! The output is truncated for brevity

Status codes: s suppressed, d damped, h history, * valid, > best

              i - internal, r RIB-failure, S stale, N Nexthop-discard

Origin codes: i - IGP, e - EGP, ? - incomplete

Prefix codes: E link, V node, T IP reacheable route, u/U unknown

              I Identifier, N local node, R remote node, L link, P prefix

              L1/L2 ISIS level-1/level-2, O OSPF, D direct, S static/peer-node

              a area-ID, l link-ID, t topology-ID, s ISO-ID,

              c confed-ID/ASN, b bgp-identifier, r router-ID,

              i if-address, n nbr-address, o OSPF Route-type, p IP-prefix

              d designated router address

   Network            Next Hop            Metric LocPrf Weight Path

* i[T][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][P[p10.0.0.11/32]]/400
```

```
                     10.0.0.11                100      0 i
* i                   10.0.0.33                100      0 i
! Further output is truncated for brevity
```

```
RP/0/0/CPU0:NPF-SR-PCE1# show bgp link-state link-state
[T][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][P[p10.0.0.11/32]]/400
Sun Apr 21 10:57:39.610 UTC
BGP routing table entry for
[T][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][P[p10.0.0.11/32]]/400
Versions:
  Process           bRIB/RIB  SendTblVer
  Speaker                 0           0
Last Modified: Apr 20 20:34:30.541 for 14:23:09
Paths: (2 available, no best path)
  Not advertised to any peer
  Path #1: Received by speaker 0
  Not advertised to any peer
  Local
    10.0.0.11 (inaccessible) from 10.0.0.11 (10.0.0.11)
      Origin IGP, localpref 100, valid, internal
      Received Path ID 0, Local Path ID 0, version 0
      Link-state: Metric: 0, PFX-SID: 11(40/0) , Extended IGP flags: 0x20
  Path #2: Received by speaker 0
  Not advertised to any peer
  Local
    10.0.0.33 (inaccessible) from 10.0.0.33 (10.0.0.33)
      Origin IGP, localpref 100, valid, internal
      Received Path ID 0, Local Path ID 0, version 0
      Link-state: Metric: 0, PFX-SID: 11(40/0) , Extended IGP flags: 0x20
```

Following the same approach as for the node NLRI and link NLRI, for the prefix NLRI, we start with the route itself. From the prefix NLRI [T][L2][I0x20][N[c65000][b0.0.0.0][s0100.0000.0011.00]][P[p10.0.0.11/32]]/400, you can glean the following information:

- **T**: This is a prefix NLRI.
- **L2**: This indicates IS-IS Level 2.
- **I0x20**: The instance-id is 32 when configured during an export on NPF-XR1 and NPF-XR3. (0x20 in hexadecimal equals 32 in decimal.)
- **N**: This indicates a local node.
- **c65000**: This is the BGP ASN of the router exporting information about this node.
- **b0.0.0.0**: This is the BGP-LS identifier (which is all zeros by default), and it can be altered in complex topologies.
- **s0100.0000.0011.00**: This is the node part of IS-IS NET without the area part.
- **P**: This is the prefix details.
- **p10.0.0.11/32**: This is the IP prefix.
- **400**: This is the overall length of the route, in bytes.

You may have noted that the local node is the same in the routes shown in Examples 16-23 through 16-25; this means the different pieces of information are related to the same node. All these routes are coming from the IS-IS LSP generated by NPF-XR1. The second part of Example 16-25 shows additional information conveyed inside the prefix SID: metric, Segment Routing prefix SID, and extended flags. To map this information to the IS-IS LSP, you can look at the IP-Extended entries as shown in Example 16-2.

In this section, you have learned how the IGP topology, including Segment Routing information, can be converted into BGP-LS routes and sent to the SDN controller, which uses these routes to build end-to-end network topology in order to provide the proper path.

Path Computation Element Protocol (PCEP)

As mentioned earlier in this chapter, the primary goal of PCEP is to program end-to-end MPLS LSPs, taking into account various constraints, such as latency, affinity, and path disjointedness. Earlier in this chapter, in the section "Segment Routing Traffic Engineering," you saw that MPLS LSPs can be configured and calculated locally on every edge router without a central entity that has an end-to-end view of the MPLS domain. However, this solution isn't scalable, as it would require you to manage probably hundreds of endpoints separately. An additional drawback is that each router can see only its part of the routing domain (for example, a single area for OSPF or level for IS-IS); therefore, a router simply can't calculate the path correctly. The approach with a centralized SDN controller aims to overcome these limitations by offloading the computation tasks to the SDN controller. Given that the SDN controller has already learned topology over BGP-LS, there is a need for a protocol to communicate between the router that wants to build an MPLS tunnel and the SDN controller that can compute the path for the MPLS

tunnel. That is where PCEP comes in.

PCEP is a protocol used for communication between a PCE (path computation element) and a PCC (path communication client), as shown in Figure 16-12.

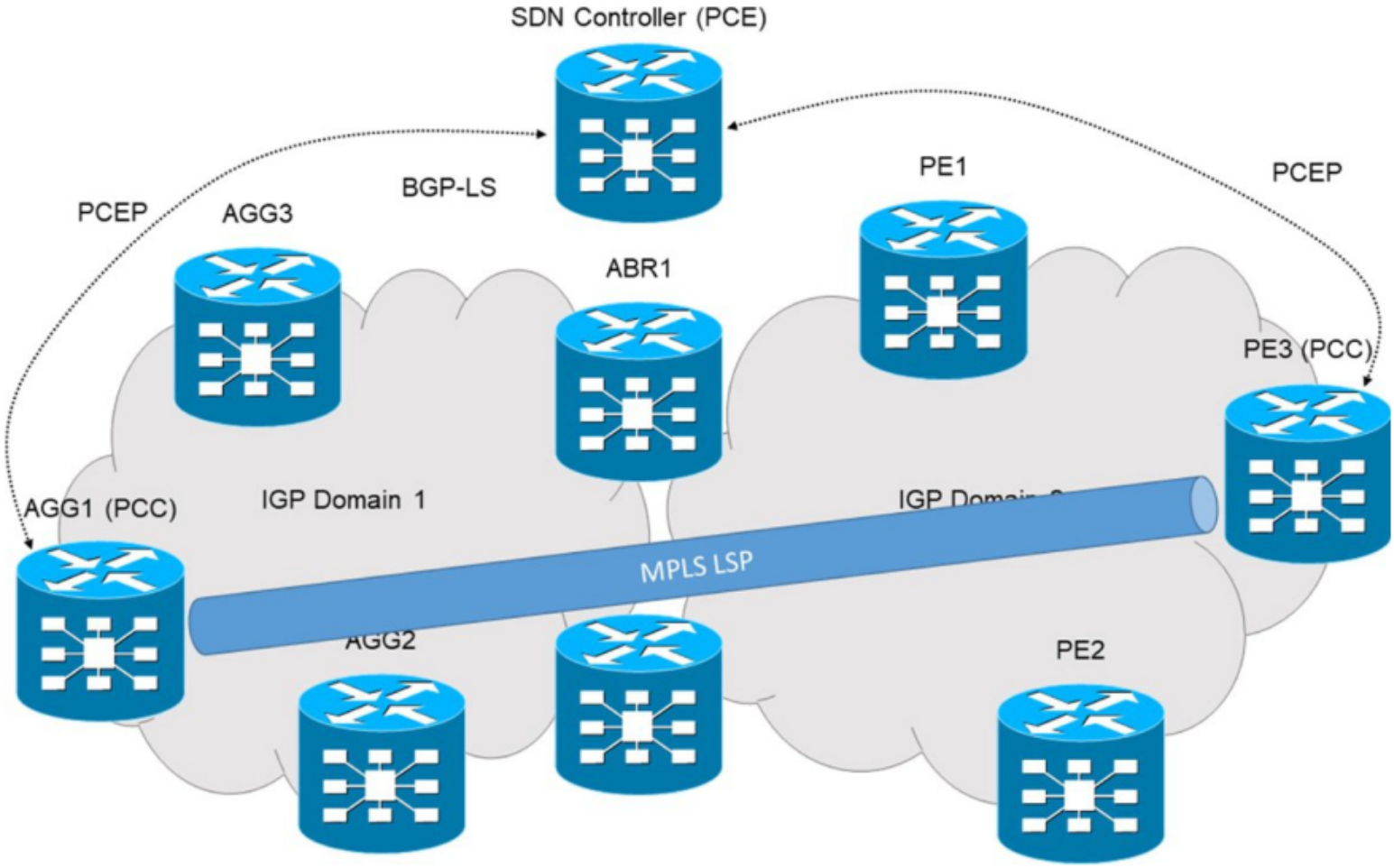

Figure 16-12 *High Level PCEP Architecture*

In Figure 16-12, PCC is an edge router that creates MPLS tunnels (the tunnel head end) but doesn't calculate the path. PCE is a function that makes the part of an SDN controller that can calculate the end-to-end MPLS path within a single MPLS domain or across multiple domains and provide this information to PCC.

There are many RFCs covering PCEP, and these are the most important of them:

• RFC 4655, "A Path Computation Element (PCE)-Based Architecture"

• RFC 5440, "Path Computation Element (PCE) Communication Protocol (PCEP)"

• RFC 8231, "Path Computation Element Communication Protocol (PCEP) Extensions for Stateful PCE"

• RFC 8281, "Path Computation Element Communication Protocol (PCEP) Extensions for PCE-Initiated LSP Setup in a Stateful PCE Model"

RFC 4655 describes general requirements and approaches to PCE deployment, including various architecture scenarios. RFC 5440 covers packet flow for the PCEP interactions, as well as the PCEP message format and their and PCEP operation in a traditional deployment, where the PCE doesn't control the MPLS LSP after initial setup. RFCs 8231 and 8281 define the extensions to PCEP for new networks, where the PCE can control the MPLS LSPs throughout its lifecycle (not only through the initial calculation and setup).

Before we get into the details of PCEP call flow, there is one more concept you should learn. Within the PCEP architecture, there are two possible ways an MPLS LSP can be established:

• **PCC initiated**: In this approach, the PCC (an ingress router) asks the PCE (an SDN controller) to calculate the path to the specific egress router. Typically, this is triggered by the configuration of the MPLS LSP on PCC with information that the PCE should use for path calculation. Two options possible:

• **With delegation**: The PCE takes further control of the MPLS LSP.

• **Without delegation**: The PCE only calculates the path and doesn't take part in further MPLS LSP control.

• **PCE initiated**: With this approach, the PCE (the SDN controller) sends the order to the PCC (the ingress router) to create an MPLS LSP to the particular egress router. This is triggered by the creation of the proper configuration (SR policy) on the SDN controller.

The first mechanism is a more traditional one in terms of configuring the endpoint, and the SDN controller just does some enhancement. The second mechanism is more scalable and fits better with network programmability, as the MPLS LSP is really programmed from the central entity.

Typical PCEP Call Flow

PCEP is a connection-oriented protocol, and to a certain extent, its session-handling capabilities are similar to those of BGP. For instance, PCEP uses TCP as the transport to build the session between a PCE and a PCC, much as BGP does. The TCP port for PCEP is 4189.

Another important factor is the way a PCEP session is established. As you might have noticed in Figure 16-12, there can be many PCCs in a network (for example, all the PE routers) and fewer PCEs. This is why a PCC is configured with the PCE's IP address as the destination; the PCC initiates a PCEP session toward the PCE, and the PCE only listens for incoming connections on the defined IP address and TCP port

4189. Figure 16-13 illustrates the PCEP session establishment process.

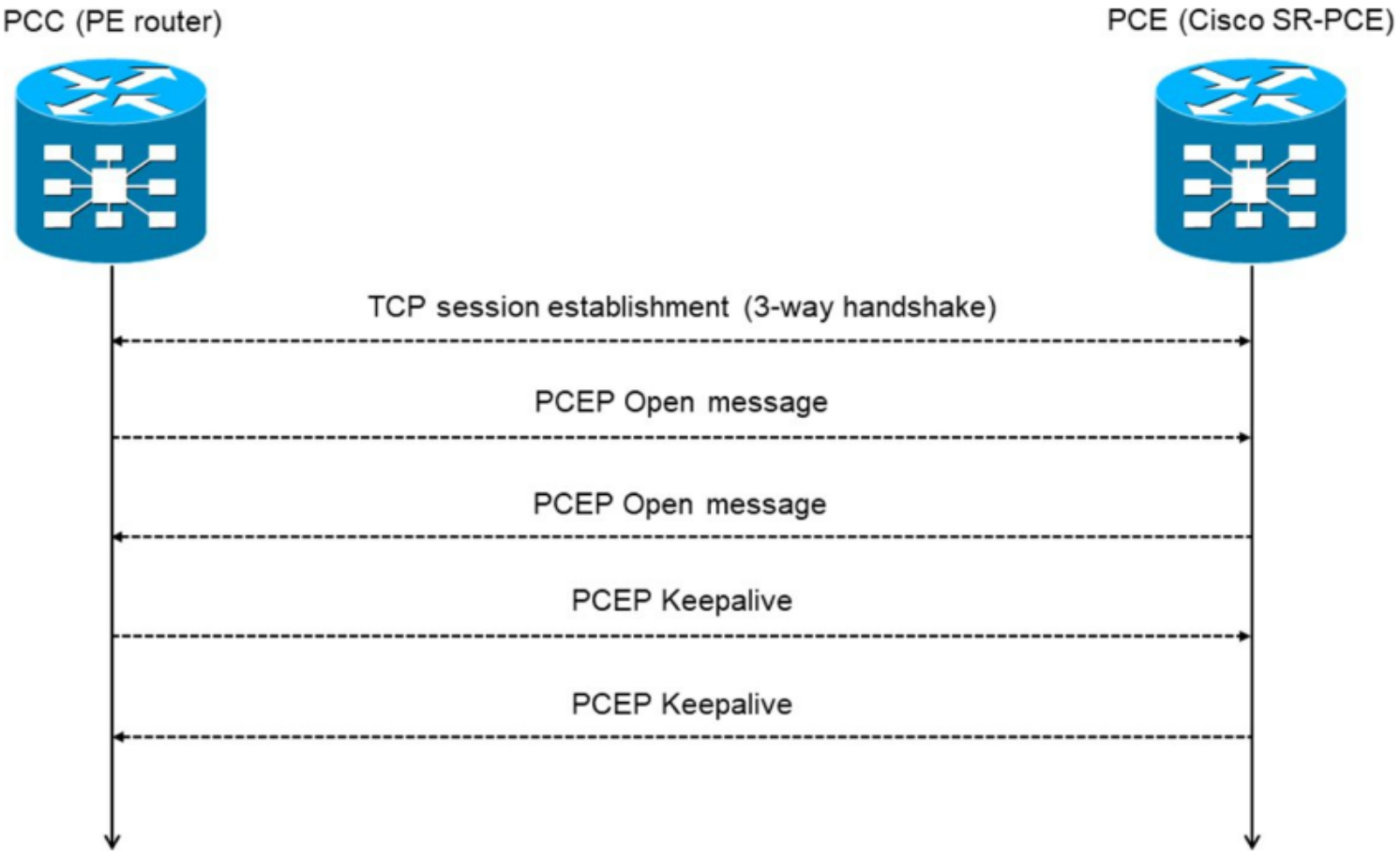

Figure 16-13 *PCEP Session Establishment*

The first step of the PCEP session establishment process is to establish the TCP session on port 4189, where the PCC initiates the PCEP session toward the PCE.

Once the TCP session is established, the PCC sends an PCEP Open message—which contains a version of PCEP, keepalive/dead timers, SID, flags, and capabilities—to the PCE, and the PCE sends its own Open message to the PCC. Much as in BGP session establishment, the transmitting/receiving process with the PCEP Open messages is asynchronous, which means it could also be the case that the PCE sends an Open message earlier than the PCC, as the source/destination TCP ports for a PCEP session are known from the initial three-way handshake process. Actually, it isn't vital who sends the Open message first. However, what is essential is that the network element must acknowledge receipt of the PCEP Open message with a PCEP Keepalive message. After both the PCE and PCC receive the PCEP Keepalive messages in response to the PCEP Open messages they sent, the PCEP session is moved to an up state.

RFC 5440 states that it is not mandatory for PCEP Keepalive messages to be sent regularly between the PCC and PCE. If the messages are sent, PCEP relies on them to track the reachability of the peer. If the messages aren't sent, PCEP relies on the TCP level (that is, the TCP session) to track whether the peer is alive. The particular implementation of the Keepalive mechanism is vendor dependent, but RFC 5440 recommends having PCEP Keepalive running. This RFC proposes to send a PCEP Keepalive message every 30 seconds (according to the keepalive timer) and to keep waiting for PCEP Keepalive messages to come four times longer than the keepalive timer (dead timer); therefore, the default value of the dead timer is 120 seconds. Cisco IOS XR has these values (that is, a 30-second keepalive timer and 120-second dead timer) in its PCEP implementation by default. If no PCEP Keepalive messages are received within the dead interval, the PCEP session is dropped as the PCEP peer is considered to be not alive.

There are a couple more items you need to know about the PCEP Open message before going further. The PCEP Open message contains information about the PCEP capabilities supported on the network element (both PCE and PCC). Based on the available capabilities, the PCC decides what it can request or send to the PCE and vice versa. The following are several of these capabilities:

• **Stateful**: In the PCEP Open messages sent by the PCC, this capability indicates that the PCC is willing to send the status of its MPLS LSPs to the PCE. On the other hand, if the PCE sends this capability, it instructs the PCC that the PCE wants to get MPLS LSP status updates. If this capability is negotiated, the PCC sends such updates to the PCE regularly, using PCEP report messages.

• **Instantiation**: In the PCEP Open message sent by the PCC, this capability informs the PCE that PCE-initiated MPLS LSPs are supported, and, therefore, the PCE can signal them to the PCC.

• **Update**: In the PCEP Open message sent by the PCC, this capability means that the PCC supports the delegation option; hence, the PCE may control the MPLS LSP on the PCC after it's initially signaled. Further, the PCC defines in the configuration whether the particular LSP is eligible for the delegation. Although this capability informs the PCE in general that the delegation might work with this PCC, the MPLS LSPs may not all be eligible for that.

• **Segment-Routing**: This capability means that the PCC can ask for the Segment Routing–based MPLS LSPs, and the PCE can provide the Segment Routing–based MPLS LSPs.

• **SR PCE**: This capability means that the PCC and/or PCE supports the establishment of MPLS traffic engineering tunnels using Segment Routing. This capability was defined later than the others, in RFC 8664 in December 2019.

If a PCEP session is successfully established and capabilities are negotiated, the next logical step in the PCEP process is to collect the information about the MPLS LSPs configured on the PCCs. This process, called *state synchronization*, is illustrated in Figure 16-14.

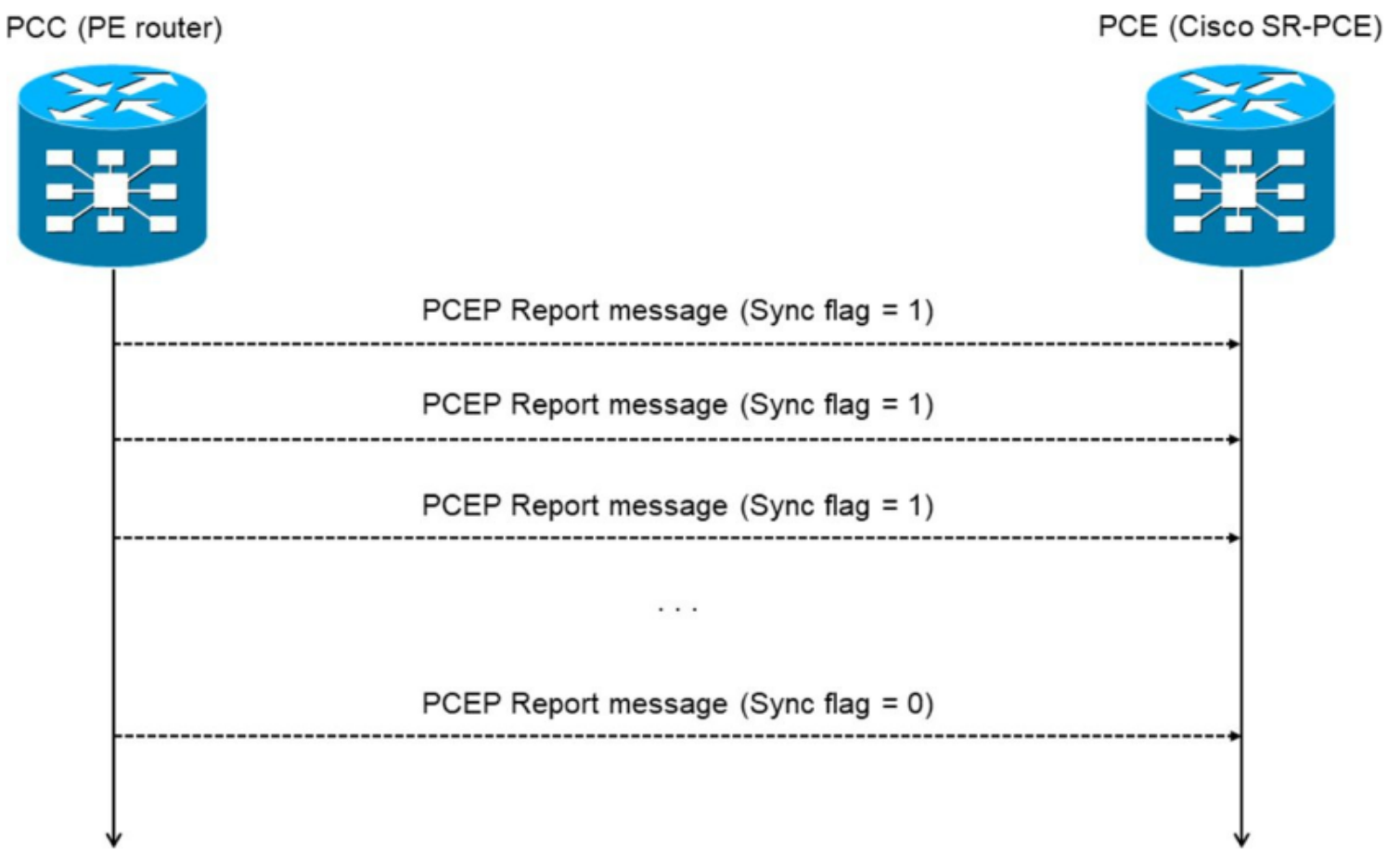

Figure 16-14 *PCEP State Synchronization*

During the state synchronization process, each PCC signals all the configured MPLS LSPs to the PCE using PCEP report messages (introduced in RFC 8231), which contain information about the LSPs (bandwidth, operational status, delegation/revocation intention, and so on). If the information about MPLS LSPs needs to be sent in several messages, then all the messages besides the last one are sent with the Sync flag value 1, and the last one is sent with the Sync flag value 0, indicating that the message is the last one. The PCE doesn't need to respond to these messages if there are no problems; in this case, the PCE is overloaded). If a problem occurs, the PCE sends back to the PCC the PCEP Error message and the proper error code (see RFC 5440 and RFC 8231).

As mentioned earlier, if the Update capability is negotiated between the PCC and PCE, the PCC can delegate control of a certain MPLS LSP to the PCE. This is usually done for inter-area scenarios because the PCE has the visibility into the whole IGP/MPLS domain or to reduce the computation efforts on the PCC. To delegate control, the PCC includes on the PCEP Report message the delegate flag set to 1 for an MPLS LSP that is to be delegated to the PCE. If the PCE accepts the delegation, it responds to the empty PCEP Update message with the MPLS LSP ID and delegation flag set to 1; if the PCE doesn't accept the delegation, it sends back the same message but with the delegation flag set to 0. Figure 16-15 shows both of these scenarios.

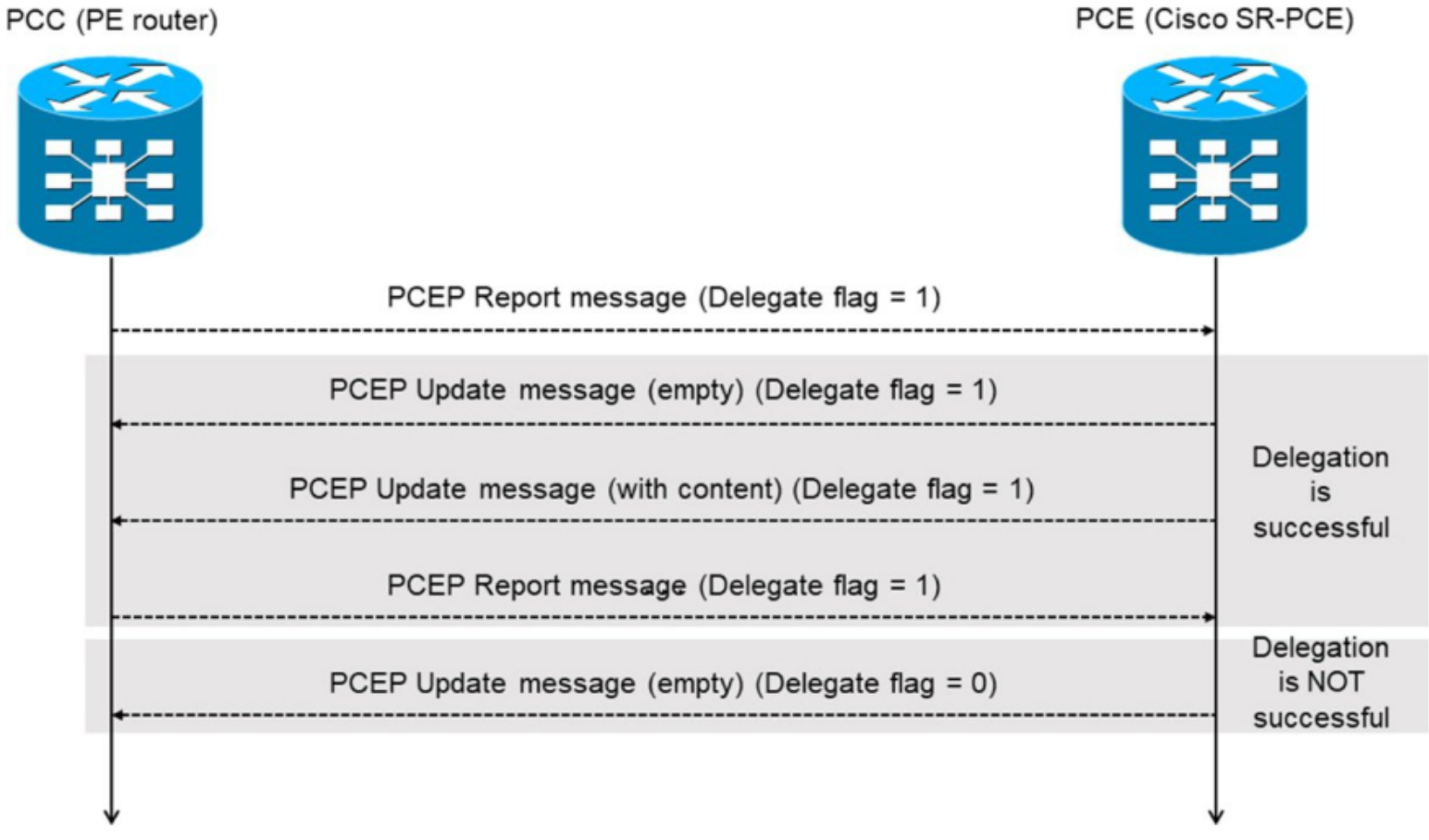

Figure 16-15 *PCEP LSP Delegation Options*

In the case of a successful delegation, the PCE is responsible for further control of the specific LSP. If the LSP needs to be changed—for example, if the network has changed and the new path should be used—the PCE sends the PCEP Update message with the proper content. The proper content is new ERO (explicit route object) containing the Segment Routing label stack for SR-TE or a set of next hops for RSVP/MPLS-TE.

The last piece of information you need to know about LSP delegation is that the PCC can revoke the delegation of a specific LSP by sending a PCEP Report message with the LSP details (LSP ID, path, and so on) and the delegate flag set to 0. Alternatively, the PCE can release the delegation by sending the PCEP Update message with the LSP details and the delegation flag set to 0. In this case, the PCC takes back control and sends the PCEP Report message back to the PCE for this LSP with the delegation set to 0.

So far, you have learned about the operation of PCEP for existing MPLS LSPs with or without delegation. Next, you need to understand how the PCEP deals with the new MPLS LSPs, and there are two primary options available.

PCEP Call Flow with Delegation

For a PCC-initiated MPLS LSP (regardless of further delegation), the PCC asks the PCE to calculate the path to the particular endpoint. This is typically triggered via configuration of the SR-TE policy or MPLS-TE tunnel on the ingress router with the PCE computation. Figure 16-16 shows a PCC-initiated MPLS LSP.

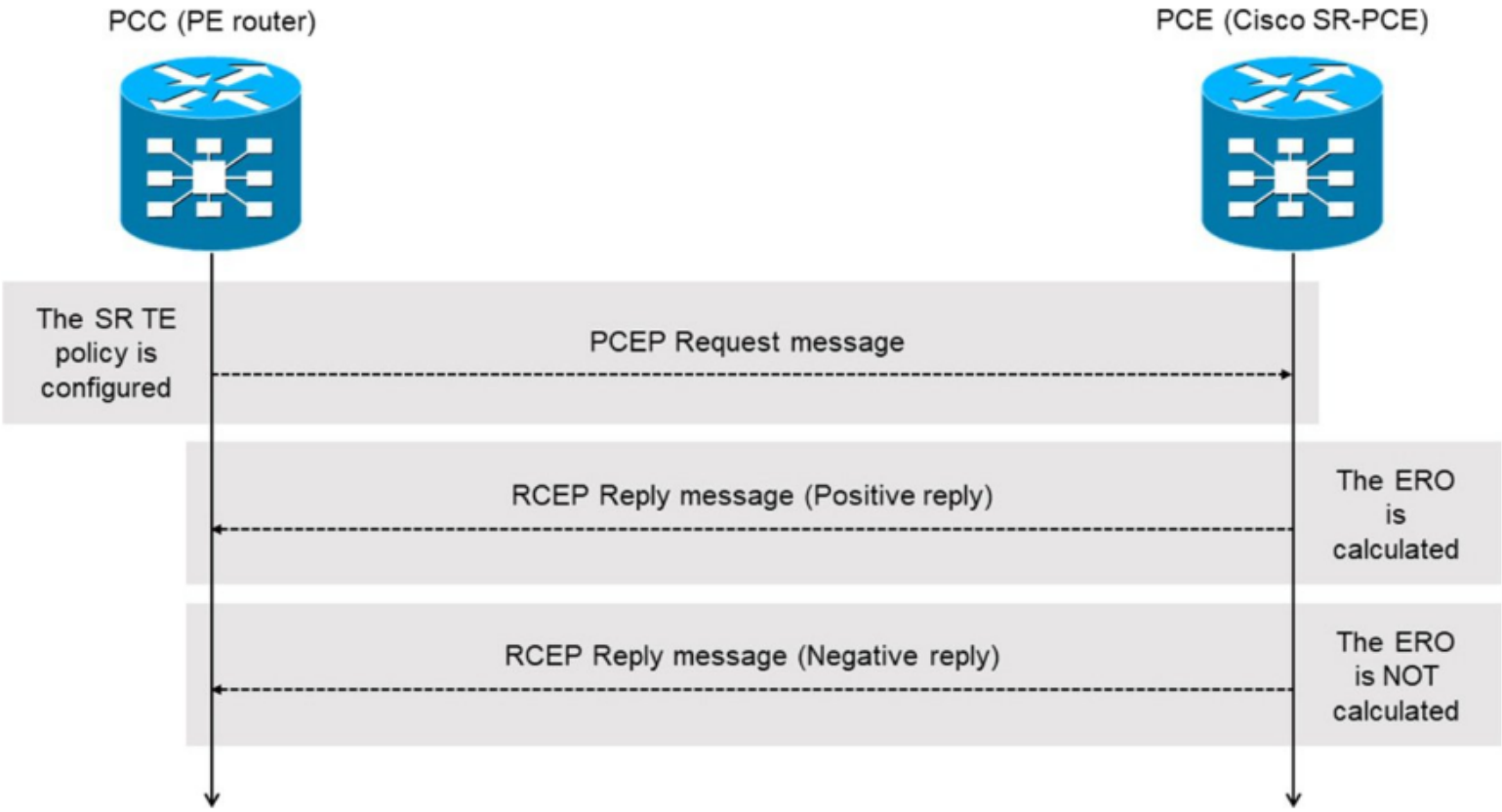

Figure 16-16 *PCC-Initiated MPLS LSP*

In Figure 16-16, you can see that the configuration of the SR-TE policy on the PCC triggers the PCEP Request message to the PCE. The PCEP Request contains the standard information for the MPLS LSP, such as the source IP address of the PCC (ingress router), the destination IP address of the egress router, the LSP ID, and the constraints (metric type, affinity, SRLG, and so on). If the PCE can find the path under the given constraints, it responds to the PCC with the PCEP Reply message with the ERO for this particular MPLS LSP. Such a message is called a *positive reply* because the MPLS LSP can be set up. If the PCE can't find the path under the given constraints, it transmits back the PCEP Reply message with the information that a path cannot be found. Such a message is called a *negative reply*. After receiving the PCEP Reply, the PCC either signals the MPLS LSP and starts using it or puts it in the operational down state (in the case of a negative reply). From here, the destiny of the MPLS LSP depends on the delegation setting. If delegation is enabled for this LSP, the process described earlier and shown in Figure 16-15 takes place. If delegation isn't enabled, the PCC sends just the PCEP Report messages with the delegate flag set to 0 for each MPLS LSP change. (RFC 8231 explains the PCC-initiated LSP both with and without delegation in more detail.)

The second primary option, as mentioned previously, is a PCE-initiated LSP. It deviates from the previous one in terms of the requestor of the LSP, as you can glean from the name. In the PCE-initiated LSP, it's the PCE that configures the particular MPLS LSP in the form of the SR-TE policy (for instance) and associates it with the PCC that should implement it. Recall that the PCC must support instantiation in order for this LSP type to work. Figure 16-17 shows the message flow for a PCE-initiated LSP.

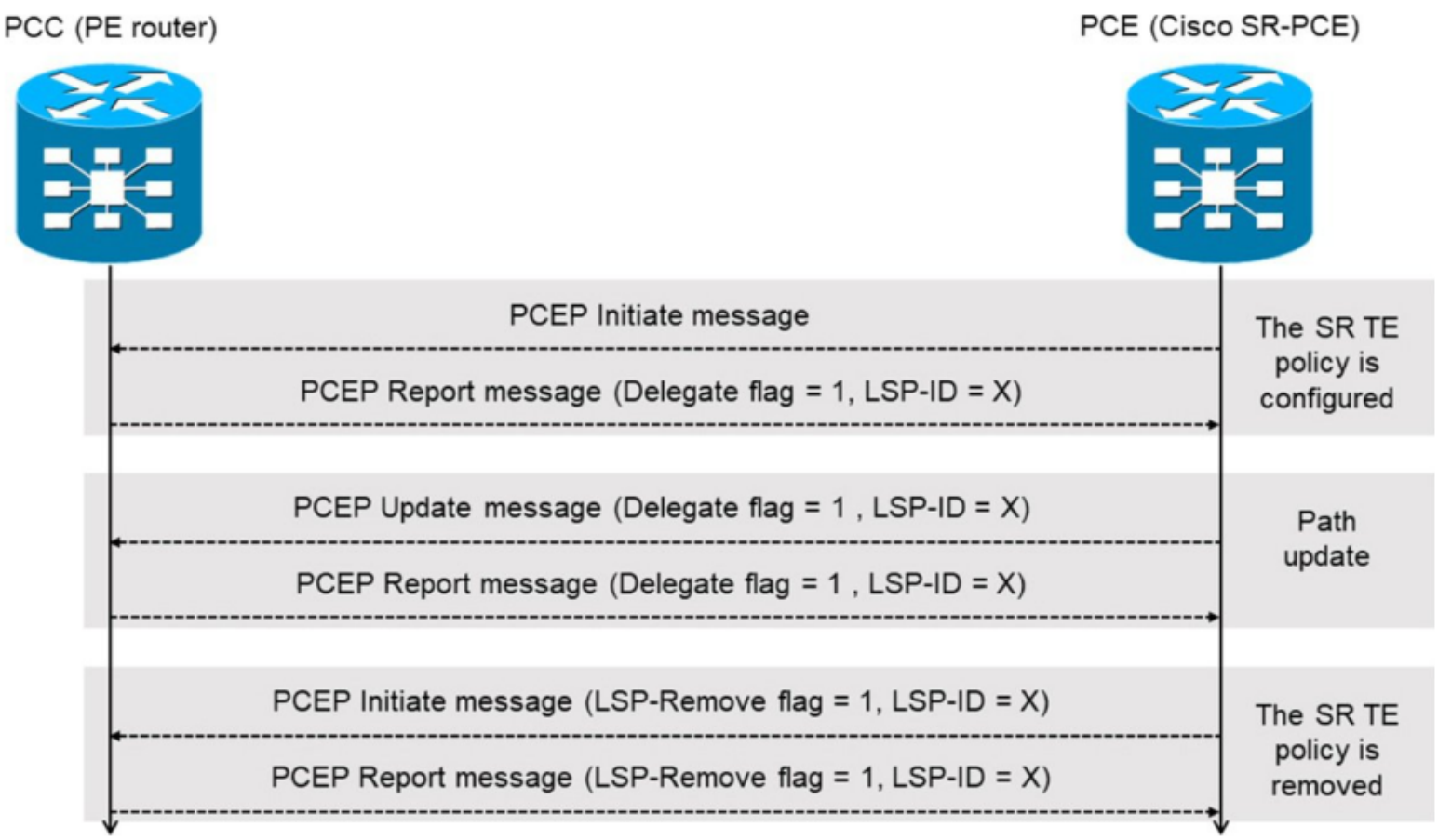

Figure 16-17 *PCE-Initiated MPLS LSP*

After the instantiation capability is negotiated between the PCC and the PCE, the PCE gets the SR-TE policy configured and associated with the particular PCC. Once the policy is configured, it triggers the PCEP to initiate message toward the PCC with the ERO and the symbolic name of the MPLS LSP. Based on this message, the PCC creates a new MPLS LSP and replies with the results to the PCE by using a PCEP Report message containing the MPLS LSP details, including the LSP ID generated locally by the PCC and the delegation flag set to 1. The LSP ID needs to be signaled back to have a unique identification of the MPLS LSP across the PCE and PCC; this is done through mapping of the symbolic SR-TE policy name and the LSP ID of the MPLS LDP as it is installed in the PCC.

Given that there are no options for "non-delegated" LSPs, the further operation of the PCE-initiated LSP type is the same as for the PCC-initiated and delegated LSPs. Any change in the IGP/MPLS topology affecting the path is calculated by the PCE and signaled using a PCEP Update message down to the PCC followed by the PCEP Report acknowledgment with the delegation flag set to 1 in both messages.

When the MPLS LSP isn't needed anymore, the SR-TE policy is either removed from the PCE configuration or put in the shutdown state. This action triggers the PCEP Initiate message for the certain MPLS LSP with the LSP-Remove flag (R flag) set to 1 and all the details about the MPLS LSP, including LSP ID. After receiving this message, the PCC removes the MPLS LSP from its forwarding table and confirms the deletion with a PCEP Report message with the LSP details and the R flag set to 1. (RFC 8281 provides details on the PCE-initiated MPLS LSPs.)

Configuring PCEP in Cisco IOS XR

To describe the configuration of PCEP on a Cisco IOS XR router, this section uses the same topology used earlier in this chapter. The configuration in this section relies on the previous configuration of IS-IS, Segment-Routing, and BGP-LS, so you need to be familiar with the earlier parts of this chapter. Figure 16-18 shows the details of the PCEP peering for this section.

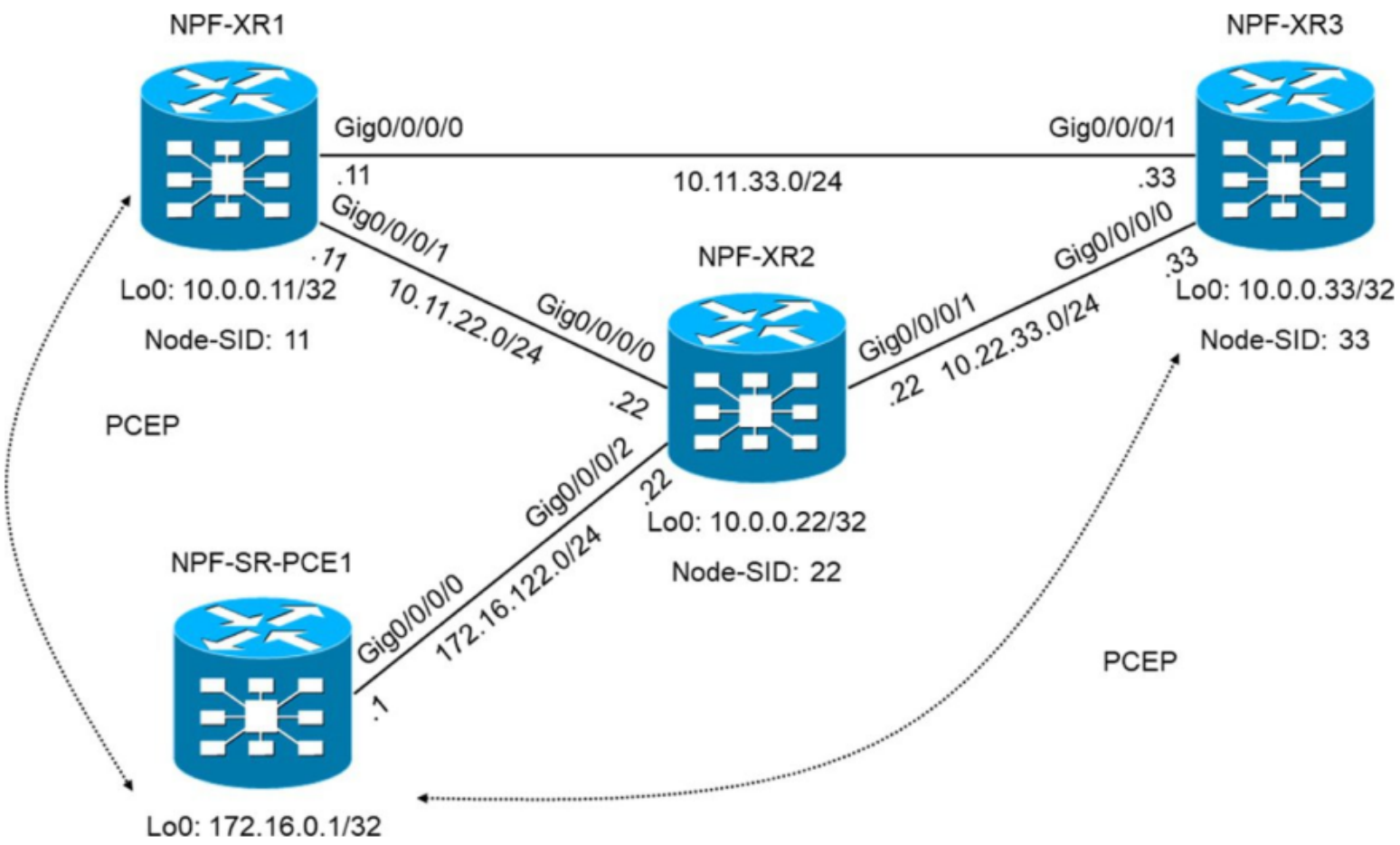

Figure 16-18 *The Lab Topology with the PCEP Lab Peering*

As you can see, PCEP follows the BGP-LS peering. (In the real world, you are most likely to have a PCEP session with each PE router, and BGP-LS will be done only with ABRs.) Based on the PCEP naming convention, PCCs refer to the client nodes: NPF-XR1 and NPF-XR3 are PCCs, whereas NPF-SR-PCE is a PCE (SDN controller).

This chapter focuses on service provider programmability based on Segment Routing, so only configuration related to Segment Routing is provided. It is important to keep this in mind because the configuration context for PCEP on the PCC depends on whether MPLS-TE/RSVP or SR-TE is used. Example 16-26 shows the configuration details for PCEP on the PCC. (Only the configuration for NPF-XR1 is shown because NPF-XR3 is configured the same way.)

Example 16-26 *Configuring the PCC for SR-TE*

```
RP/0/0/CPU0:NPF-XR1(config)#show conf
Sun May 12 15:33:33.620 UTC
Building configuration...
!! IOS XR Configuration 6.5.1
segment-routing
 traffic-eng
  pcc
   source-address ipv4 10.0.0.11
   pce address ipv4 172.16.0.1
    password encrypted 002A2320
   !
   report-all
  !
 !
!
end
```

You can see in Example 16-26 that the configuration is done within the **segment-routing traffic-eng** context. You need to specify the local (PCC) and remote (PCE) addresses and the password for enhanced security (optionally), and you need to instruct the PCC to report all the available MPLS LSPs to the PCE.

The configuration of the PCE is done within another configuration context, as shown in Example 16-27.

Example 16-27 *Configuring the PCC for SR-TE*

```
RP/0/0/CPU0:NPF-SR-PCE1(config)#show conf
Sun May 12 15:39:17.471 UTC
pce
```

```
 address ipv4 172.16.0.1
 password encrypted 100A1C0956052D
 segment-routing
  traffic-eng
  !
 !
!
end
```

The **pce** configuration knob is standard for both SR-TE and MPLS-TE. You can see in Example 16-27 that there is only an IP address configured, where the PCE listens for incoming sessions. There are no other PCEP peers configured, as they are initiated on the PCC. It's crucial that the IP address configured on the PCE (**address ipv4 172.16.0.1** in Example 16-27) match the peer address on the PCC (**pce address ipv4 172.16.0.1** in Example 16-26); otherwise, PCEP sessions will not be established.

If you have configured both the PCE and the PCC correctly, the sessions follow the process shown earlier, and you can verify the state of the PCEP peering. On the PCC, you check the PCEP session state as shown in Example 16-28.

Example 16-28 *Verifying the PCEP Peering from the PCC*

```
RP/0/0/CPU0:NPF-XR1#show segment-routing traffic-eng pcc ipv4 peer detail
Sun May 12 15:46:35.656 UTC

PCC's peer database:
--------------------

Peer address: 172.16.0.1 (best PCE)
  State up
  Capabilities: Stateful, Update, Segment-Routing, Instantiation
  PCEP has been up for: 04:18:44
  Local keepalive timer is 30 seconds
  Remote keepalive timer is 30 seconds
  Local dead timer is 120 seconds
  Remote dead timer is 120 seconds
  Statistics:
    Open messages:      rx 2        |  tx 2
    Close messages:     rx 0        |  tx 1
    Keepalive messages: rx 674      |  tx 677
    Error messages:     rx 0        |  tx 0
    Report messages:    rx 0        |  tx 23
    Update messages:    rx 0        |  tx 0
```

In the output shown in Example 16-28, you see the state of the session, which is up. You see also the capabilities negotiated between the PCC and PCE, followed by the parameters of the keepalive/dead timers and statistics of the different PCEP messages transmitted and received.

From the PCE's perspective, you use another command that is related to the PCE configuration context, as shown in Example 16-29.

Example 16-29 *Verifying the PCEP Peering from the PCE*

```
RP/0/0/CPU0:NPF-SR-PCE1#show pce ipv4 peer detail
Sun May 12 15:51:31.121 UTC

PCE's peer database:
--------------------

Peer address: 10.0.0.11
  State: Up
  Capabilities: Stateful, Segment-Routing, Update, Instantiation
  PCEP has been up for: 04:23:42
  PCEP session ID: local 0, remote 0
  Sending KA every 30 seconds
  Minimum acceptable KA interval: 20 seconds
  Peer timeout after 120 seconds
  Maximum SID Depth: 16
  Statistics:
```

```
    Keepalive messages: rx   526 tx   526
    Request messages:   rx     0 tx     0
    Reply messages:     rx     0 tx     0
    Error messages:     rx     0 tx     0
    Open messages:      rx     1 tx     1
    Report messages:    rx    11 tx     0
    Update messages:    rx     0 tx     0
    Initiate messages:  rx     0 tx    10
  Last PCError:
    Received: None
    Sent: None


Peer address: 10.0.0.33
  State: Up
  Capabilities: Stateful, Segment-Routing, Update, Instantiation
  PCEP has been up for: 04:23:42
  PCEP session ID: local 0, remote 0
  Sending KA every 30 seconds
  Minimum acceptable KA interval: 20 seconds
  Peer timeout after 120 seconds
  Maximum SID Depth: 16
  Statistics:
    Keepalive messages: rx   526 tx   527
    Request messages:   rx     0 tx     0
    Reply messages:     rx     0 tx     0
    Error messages:     rx     0 tx     0
    Open messages:      rx     1 tx     1
    Report messages:    rx     7 tx     0
    Update messages:    rx     0 tx     0
    Initiate messages:  rx     0 tx     4
  Last PCError:
    Received: None
    Sent: None
```

On the PCE, you see the details for both PCCs connected to NPF-SR-PCE (that is, NPF-XR1 and NPF-XR3). The content in Examples 19-28 and 16-29 is quite similar, although there is a bit more information on the PCE-related maximum SID label depth of the ERO for SR-TE, and there are more packet types.

In Example 16-26, the PCC is instructed to send all the local SR-TE LSPs to the PCE. At the beginning of this chapter, you saw the configuration of the SR-TE policies on NPF-XR1 and NPF-XR3. Based on the fact that the counter of the received PCEP Report messages on the PCE isn't zero (refer to Example 16-29), you can assume that the PCCs send information about the LSPs to the PCE. This is indeed the case in this instance, and you can review these MPLS LSPs on the PCE as shown in Example 16-30.

Example 16-30 *Verifying the Reported MPLS LSPs on the PCE*

```
RP/0/0/CPU0:NPF-SR-PCE1#show pce lsp detail
Sun May 12 16:27:54.761 UTC


PCE's tunnel database:
----------------------
PCC 10.0.0.11:


Tunnel Name: cfg_XR1_XR3_OVER_XR2_BLUE
 LSPs:
  LSP[0]:
   source 10.0.0.11, destination 10.0.0.33, tunnel ID 1, LSP ID 3
   State: Admin up, Operation active
   Setup type: Segment Routing
   Binding SID: 1000002
```

```
   Maximum SID Depth: 16
   Bandwidth: signaled 0 kbps, applied 0 kbps
   PCEP information:
     PLSP-ID 0x80001, flags: D:0 S:0 R:0 A:1 O:2 C:0
   LSP Role: Single LSP
   State-sync PCE: None
   PCC: 10.0.0.11
   LSP is subdelegated to: None
   Reported path:
     Metric type: IGP, Accumulated Metric 0
      SID[0]: Node, Label 16022, Address 10.0.0.22
      SID[1]: Unknown, Label 16033,
   Computed path: (Local PCE)
     None
     Computed Time: Not computed yet
   Recorded path:
     None
   Disjoint Group Information:
     None


PCC 10.0.0.33:


Tunnel Name: cfg_XR3_XR1_OVER_XR2_BLUE
 LSPs:
  LSP[0]:
   source 10.0.0.33, destination 10.0.0.11, tunnel ID 1, LSP ID 3
   State: Admin up, Operation active
   Setup type: Segment Routing
   Binding SID: 1000001
   Maximum SID Depth: 16
   Bandwidth: signaled 0 kbps, applied 0 kbps
   PCEP information:
     PLSP-ID 0x80001, flags: D:0 S:0 R:0 A:1 O:2 C:0
   LSP Role: Single LSP
   State-sync PCE: None
   PCC: 10.0.0.33
   LSP is subdelegated to: None
   Reported path:
     Metric type: IGP, Accumulated Metric 0
      SID[0]: Adj, Label 24003, Address: local 10.22.33.33 remote 10.22.33.22
      SID[1]: Unknown, Label 24001,
   Computed path: (Local PCE)
     None
     Computed Time: Not computed yet
   Recorded path:
     None
   Disjoint Group Information:
     None
```

In Example 16-30 you can see all the MPLS LSPs currently installed in the network. The output is quite extensive and includes the symbolic tunnel name, source/destination IP addresses, the tunnel ID, the LSP ID, the PCEP PLSP-ID (because the PCC creates a unique PLSP-ID for each LSP that is constant for the lifetime of a PCEP session), the status of the delegation, the reported path (that is, RRO [recorded route object]), and more. You see also the status of those LSPs, which are admin and operational up.

Now we have reached the point where the PCE should perform the computation of the MPLS LSP; it can be either a PCC-initiated or PCE-initiated operation. The prerequisites for these computations are the routing and segment routing information. There are two ways to distribute this information: either by using IGP, where the PCE must be part of the IGP domain, or by using BGP-LS. The BGP-LS peering between the

PCE and the PCCs is configured earlier in this chapter (refer to Example 16-17), and that information is used to populate the PCE. Actually, no additional configuration is required, and you can check the outcome of the topology used by PCEP as shown in Example 16-31.

Example 16-31 *Verifying the Routing Topology Used by the PCE*

```
RP/0/0/CPU0:NPF-SR-PCE1#show pce ipv4 top

Sun May 12 16:45:32.039 UTC


PCE's topology database - detail:

--------------------------------

Node 1

  Host name: NPF-XR1

  ISIS system ID: 0100.0000.0011 level-2 ASN: 65000 domain ID: 32

  Prefix SID:

    ISIS system ID: 0100.0000.0011 level-2 ASN: 65000 domain ID: 32

      Prefix 10.0.0.11, label 16011 (regular), flags: N

  SRGB INFO:

    ISIS system ID: 0100.0000.0011 level-2 ASN: 65000 domain ID: 32

      SRGB Start: 16000 Size: 8000


  Link[0]: local address 10.11.22.11, remote address 10.11.22.22

    Local node:

      ISIS system ID: 0100.0000.0011 level-2 ASN: 65000 domain ID: 32

    Remote node:

      Host name: NPF-XR2

      ISIS system ID: 0100.0000.0022 level-2 ASN: 65000 domain ID: 32

    Metric: IGP 10, TE 10, Latency 10

    Bandwidth: Total 0 Bps, Reservable 0 Bps

    Admin-groups: 0x00000000

    Adj SID: 24001 (unprotected)


  Link[1]: local address 10.11.33.11, remote address 10.11.33.33

    Local node:

      ISIS system ID: 0100.0000.0011 level-2 ASN: 65000 domain ID: 32

    Remote node:

      Host name: NPF-XR3

      ISIS system ID: 0100.0000.0033 level-2 ASN: 65000 domain ID: 32

    Metric: IGP 10, TE 10, Latency 10

    Bandwidth: Total 0 Bps, Reservable 0 Bps

    Admin-groups: 0x00000000

    Adj SID: 24003 (unprotected)


Node 2

  Host name: NPF-XR2

! Further output is truncated for brevity
```

When you have verified that the routing information is available on the PCE, the last step is to create the MPLS LSPs.

Note

SR-PCE (a Cisco IOS XR–based PCE) is actively being developed as this book goes to press. During the writing of the book, the Cisco IOS XR versions 6.5.1, 6.5.2, and 6.5.3 were used. It is important to note that not all combinations of IOS XR on the PCC and PCE work successfully. Additional complexity comes from the fact that the behavior of the PCE on the IOS XRv9000 (production SR-PCE) and IOS XRv used in VIRL is a bit different. If something isn't working in your setup, try to use another software version. We hope that the SR-PCE will be stabilized in the Cisco IOS XR 7.* software release.

As explained previously, the PCC-initiated MPLS LSP is configured on the PCC, and then the PCE calculates the path. Example 16-32 shows a sample configuration done on the PCC NPF-XR1.

Example 16-32 *Configuring the PCC-Initiated LSP on the PCC*

```
RP/0/0/CPU0:NPF-XR1(config)#show conf
```

```
Sun May 12 19:56:33.099 UTC
Building configuration...
!! IOS XR Configuration 6.5.1.34I
segment-routing
 traffic-eng
  policy XR1_XR2_PCC
   binding-sid mpls 1000012
   color 100 end-point ipv4 10.0.0.33
   candidate-paths
    preference 10
     dynamic
      pcep
      !
      metric
       type igp
      !
     !
    !
   !
  !
 !
!
end
```

The content of this configuration is explained earlier in this chapter, in the explanation of Example 16-9. The only difference here is that **candidate-path** is **dynamic**, and **pcep** should be used (so that the PCC asks the PCE to calculate the path). After you have configured the SR-TE policy, it should be visible locally on the PCC, as shown in Example 16-33.

Example 16-33 *Verifying the PCC-Initiated LSP on the PCC*

```
RP/0/0/CPU0:NPF-XR1#show segment-routing traffic-eng policy name XR1_XR2_PCC
Sun May 12 20:03:37.260 UTC


SR-TE policy database
---------------------


Name: XR1_XR2_PCC (Color: 100, End-point: 10.0.0.33)
  Status:
    Admin: up  Operational: up for 00:06:22 (since May 12 19:57:14.506)
  Candidate-paths:
    Preference 10:
      Dynamic (pce 172.16.0.1) (active)
        Metric Type: IGP,   Path Accumulated Metric: 10
          16033 [Prefix-SID, 10.0.0.33]
  Attributes:
    Binding SID: 1000012
      Allocation mode: explicit
      State: Programmed
      Policy selected: yes
    Forward Class: 0
    Steering BGP disabled: no
    IPv6 caps enable: no
```

In the details of this MPLS LSP, you can see that it's dynamic, and the PCE calculated the path (including the IP addresses of the PCE defining which PCE). Because the metric type IGP was defined, you can see it in the provided output, along with the calculated IGP metric of the whole MPLS LSP.

Because all the SR-TE LSPs are reported to the PCE according to the configuration for this chapter, you should be able to view the details on the PCE as well (see Example 16-34).

Example 16-34 *Verifying the PCC-Initiated LSP on the PCE*

```
RP/0/0/CPU0:NPF-SR-PCE1#show pce lsp name cfg_XR1_XR2_PCC detail

Sun May 12 20:04:16.541 UTC


PCE's tunnel database:

----------------------

PCC 10.0.0.11:


Tunnel Name: cfg_XR1_XR2_PCC

 LSPs:

  LSP[0]:

   source 10.0.0.11, destination 10.0.0.33, tunnel ID 6, LSP ID 2

   State: Admin up, Operation active

   Binding SID: 1000012

   Maximum SID Depth: 16

   Bandwidth: signaled 0 kbps, applied 0 kbps

   PCEP information:

     PLSP-ID 0x80006, flags: D:1 S:0 R:0 A:1 O:0 C:0

   LSP Role: Single LSP

   State-sync PCE: None

   PCC: 10.0.0.11

   LSP is subdelegated to: None

   Reported path:

     Metric type: IGP, Accumulated Metric 10

      SID[0]: Node, Label 16033, Address 10.0.0.33

   Computed path: (Local PCE)

     Computed Time: Sun May 12 19:57:13 UTC 2019 (00:07:03 ago)

     Metric type: IGP, Accumulated Metric 10

      SID[0]: Node, Label 16033, Address 10.0.0.33

   Recorded path:

     None

   Disjoint Group Information:

     None
```

As highlighted in Example 16-34, the PCEP flags are marked up. The D flag stands for delegation, meaning that the control for this LSP is delegated from the PCC to the PCE. You can also see the details of the path computation, as well as information reported by the PCC. This information (Computed path and Reported path) should be equal to each other in a stable network, whereas during convergence, it might be unequal if the PCE has computed the new path and sent it to the PCC but the PCC hasn't yet installed it and therefore hasn't sent back the PCEP Report message. You might also note that all the LSPs configured on the PCC and signaled to the PCE have the **cfg_** prefix in their symbolic names.

The second type of MPLS LSP where the PCE is involved is the PCE-initiated LSP. As indicated by the name and as explained earlier in this chapter, the PCE initiated the MPLS LSP, which is why the proper SR-TE policy is configured on the PCE. Example 16-35 shows the details of this type of configuration.

Example 16-35 *Configuring the PCE-Initiated LSP on the PCC*

```
RP/0/0/CPU0:NPF-SR-PCE1(config)#show conf

Sun May 12 20:34:28.687 UTC

Building configuration...

!! IOS XR Configuration 6.5.1

pce

 segment-routing

  traffic-eng

   peer ipv4 10.0.0.33

    policy PCE_XR3_XR1

     binding-sid mpls 1000021

     color 100 end-point ipv4 10.0.0.11

     candidate-paths

      preference 20

       dynamic mpls
```

```
        metric
         type igp
        !
       !
      !
     !
    !
   !
  !
 !
!
end
```

This is the first time you have seen the PCEP peer configuration on the PCE. This configuration is necessary because the PCE needs to identify where to send the particular PCE Initiate message with the LSP details. Otherwise, the configuration of the SR-TE policy is absolutely the same as for the PCC (refer to Example 16-9).

The verification process for the PCE-initiated LSP is different from the verification process for the PCC-initiated LSP. In this case, you start with the PCE, as that is where the configuration took place. The command to obtain the necessary information is the same as for the PCC, as you can see in Example 16-36.

Example 16-36 *Verifying the PCE-Initiated LSP on the PCE*

```
RP/0/0/CPU0:NPF-SR-PCE1#show pce lsp name PCE_XR3_XR1 detail
Sun May 12 21:07:38.070 UTC


PCE's tunnel database:
----------------------
PCC 10.0.0.33:


Tunnel Name: PCE_XR3_XR1
 LSPs:
  LSP[0]:
   source 10.0.0.33, destination 10.0.0.11, tunnel ID 5, LSP ID 2
   State: Admin up, Operation active
   Binding SID: 1000021
   Maximum SID Depth: 16
   Bandwidth: signaled 0 kbps, applied 0 kbps
   PCEP information:
     PLSP-ID 0x80005, flags: D:1 S:0 R:0 A:1 O:0 C:1
   LSP Role: Single LSP
   State-sync PCE: None
   PCC: 10.0.0.33
   LSP is subdelegated to: None
   Reported path:
     Metric type: IGP, Accumulated Metric 10
      SID[0]: Node, Label 16011, Address 10.0.0.11
   Computed path: (Local PCE)
     Computed Time: Sun May 12 20:46:54 UTC 2019 (00:20:44 ago)
     Metric type: IGP, Accumulated Metric 10
      SID[0]: Node, Label 16011, Address 10.0.0.11
   Recorded path:
     None
   Disjoint Group Information:
     None
```

Because no constraints were set, in Example 16-36, the PCE calculates the path following the lowest IGP end-to-end metrics; therefore, the accumulated metric is equal to the metric you could find in the routing table. From the flag's perspective, it's a bit different from Example 16-34 because the LSP type is different, but the delegation flag is also set to 1, as in Example 16-34. Example 16-37 shows output from the router's (that is, the PCC's) perspective.

Example 16-37 *Verifying the PCE-Initiated LSP on the PCC*

gNMI."

Network Platforms

This section covers the programmability and automation features of the three major Cisco software trains that run most of Cisco's networking platforms: Open NX-OS, IOS XE, and IOS XR. It covers four use cases: Linux shells and NETCONF on each of the three platforms and NX-API CLI and NX-API REST on Nexus switches.

Networking APIs

This section describes, at a high level, the programmability and automation capabilities of each of the three Cisco network operating systems.

Open NX-OS Programmability

Open NX-OS is Cisco's network operating system for the programmable models of the Nexus line of switches, which are commonly used in data center networks. Open NX-OS is based on a customized version of the Wind River Linux distro. The Linux Foundation started the Yocto project in 2010 to customize Linux distros for embedded systems, and the Wind River Linux distro is one of them.

Open NX-OS runs as an application on top of that Linux distro. In addition, an NGNIX HTTP server runs, also as an application on top of Linux, and handles all communications with the switches that use HTTP/HTTPS. The Linux OS and the NGNIX HTTP server provide a foundation for many of the programmable NX-OS interfaces.

Note

For the sake of brevity, moving forward, Open NX-OS will simply be referred to as NX-OS only.

The following is a non-comprehensive list of programmable interfaces available for managing a Nexus switch running NX-OS:

• **Bash and Guest shell access**: Programmable Nexus switches expose two on-box Linux shells that can be used to manage a Nexus switch in a fashion similar to managing a Linux server. The first, the Bash shell, provides access to the actual Linux OS on which NX-OS is running as an application, and the second, the Guest shell provides access to a separate execution space that is decoupled from NX-OS. The Guest shell runs in a Linux Containers (LXC) container.

• **NX-API CLI**: This API enables a switch to accept CLI and Bash commands encoded in JSON or XML, execute those commands on box, and then send back the results, encoded in either JSON or XML. The use of the NX-API CLI is made easier through the Developer Sandbox, a web application that runs on a Nexus switch and provides several facilities for testing this API, including converting CLI commands into XML- and JSON-encoded API calls. The Developer Sandbox can also send these calls to the switch and display the responses.

• **NX-API REST**: This RESTful API can be used to fully manage a Nexus switch.

• **Model-based industry-standard APIs**: NX-OS can expose NETCONF, RESTCONF, gRPC, and gNMI APIs.

• **Configuration management automation**: Ansible includes modules for configuring NX-OS devices. Nexus switches have built-in support for Puppet and Chef agents.

• **OpenStack Neutron ML2 support**: Nexus switches provide a plug-in that allows an OpenStack implementation to manage a switch running NX-OS as part of a complete ecosystem of network, compute, storage, and many other components that constitute a private cloud.

• **Application hosting**: The Guest shell can be leveraged to run Python scripts on box or to install and run 64-bit Linux applications packaged in RPM format on the switch. This is facilitated by the fact that the **yum** package manager is included by default in the Guest shell. NX-OS devices also come with a preinstalled Docker daemon that is used to run on-box containers in the Bash shell on which a 64-bit Linux application may be hosted.

• **Telemetry**: NX-OS supports model-driven streaming telemetry transported using JSON over UDP or GPB over gRPC (over HTTP/2).

Note

Not all the listed interfaces are available on all Nexus switches. At the time of writing this chapter, most of these interfaces are available on most programmable Nexus switches (3000 and 9000 Series) running version 9.3 or later of Open NX-OS. Make sure to read the configuration guide, programmability guide, and release notes for the specific hardware model and software version of your platform to identify which interfaces are available for that specific platform and which are not.

IOS XE Programmability

IOS XE is Cisco's network operating system for enterprise-grade network devices. Some flavor of IOS XE runs on most enterprise routers, switches, SD-WAN, or wireless devices. IOS XE is based on a customized version of the Wind River Linux distro.

As with NX-OS, an NGNIX HTTP server runs on top of the Linux kernel and handles all HTTP/HTTPS-based communications with the devices.

The following is a non-comprehensive list of programmable interfaces available for managing a device running IOS XE (but note that not all interfaces are applicable to all hardware platforms or software versions):

• **Guest shell access**: IOS XE devices expose a containerized Linux OS shell that can be used for running on-box scripts such as Python programs. Some IOS XE devices run a full-fledged Guest shell similar to the one exposed by NX-OS where RPM packages can be installed; some IOS XE devices run a Lite version of the Guest shell that has different capabilities than the full shell. In both cases, the Guest shell runs in an LXC container on the device.

• **Model-based industry-standard APIs**: IOS XE can expose NETCONF, RESTCONF, gRPC, and gNMI APIs.

• **Configuration management automation**: Ansible includes modules for configuring IOS XE devices. IOS XE has built-in support for both Puppet and Chef agents.

• **Application hosting**: Through the IOx (IOS + Linux) framework, also called the Cisco Application Framework (CAF), IOS XE devices support on-box KVM and LXC containers on which an application may be hosted.

• **OpenFlow**: IOS XE devices support OpenFlow, and a device can be controlled from a third-party OpenFlow controller.

• **Telemetry**: IOS XE supports model-driven streaming telemetry. Subscriptions to telemetry data items can be established using NETCONF,

RESTCONF, or gNMI, and the actual telemetry stream is sent to the receivers using the YANG push mechanism over a NETCONF session.

IOS XR Programmability

IOS XR is Cisco's network operating system for the ASR, NCS, and 8000 Series routers, and it is used primarily in service provider environments. Starting with Version 6.x, IOS XR is based on a customized version of the Wind River Linux distro. Platforms running IOS XR Version 6.x run IOS XR inside an LXC container on top on the Wind River distro. The administrative plane also runs in its own LXC container. Starting with Version 7.x, IOS XR runs directly as a number of processes on top on the Wind River distro without a container.

The following is a non-comprehensive list of programmable interfaces available for managing an IOS XR–based router running IOS XR Version 7.x (but note that not all interfaces are applicable to all hardware platforms or software versions):

• **Bash shell access**: IOS XR devices expose the underlying Linux system on which the IOS XR software runs as a group of processes. IOS XR does not provide a Guest shell out of the box. You can, however, create LXC or Docker containers as required on top of the Bash shell.

• **Model-based industry-standard APIs**: IOS XR exposes three model-based APIs: NETCONF, gRPC, and gNMI.

• **Service layer APIs**: IOS XR exposes the same service layer APIs that the router's management plane uses to speak with its own control plane, which is more formally referred to as the *service adaptation layer*. These internal APIs are exposed such that you can make an API call to modify the device's control plane directly. For example, you may run your own routing protocol on a device, and this routing protocol would follow an algorithm of its own to calculate the best path to a route. This protocol may then inject routes into the RIB directly, without having to go through configuration changes that eventually reflect on the routing table entries. These routes would be labeled with an A in the routing table to signify that these are application routes, versus, for example, OSPF routes (which are labeled with an O).

• **Application hosting**: IOS XR devices come with a preinstalled Docker daemon used to run on-box containers on which an application may be hosted. The devices also support user-created LXC containers.

• **Configuration management automation**: Ansible includes modules for configuring IOS XR devices. IOS XR devices have built-in support for Puppet and Chef agents. These agents come as native RPM packages and run as processes on the Wind River distro.

• **Telemetry**: IOS XR supports model-driven streaming telemetry transported using JSON over TCP or UDP or using GPB over gRPC (over HTTP/2).

Use Cases

This section presents four use-cases of programmability and automation for network platforms. The first two cover the Bash and Guest shells and the other two discuss the NX-API CLI and the RESTful NX-API REST APIs.

Use Case 1: Linux Shells

NX-OS runs as a set of processes in the user space on top of a Wind River Linux distro. As of this writing, NX-OS Version 9.2.1 runs on Linux kernel Version 4.1. Nexus switches running NX-OS provide access to two Linux shells: the Bash shell and the Guest shell. Both shells are, technically, Bash shells.

The Bash shell provides shell access to the actual Linux OS that runs a switch. This is referred to as the *underlying* Linux OS. The Guest shell provides access to a lightweight Linux OS running in an LXC container on top of the underlying Linux OS.

Since the Bash shell provides full access to the underlying OS running the switch, you get full access to all aspects of the switch, much as you would when running Linux on a regular server. Any operation done in the Bash shell will impact the functionality of the switch.

On the other hand, the containerized Linux OS through the Guest shell runs in a semi-isolated environment and provides limited access to the switch hardware and some of the other software running on the switch, including NX-OS, which itself is a group of processes running on top of the underlying Linux OS.

The Guest shell provides an environment that allows you to run Linux-based applications without having to interact directly with the underlying Linux system that runs the box. This means that the environment through the Guest shell may be customized based on the requirements of the software you plan on running there (for example, a specific version of Python). In addition, whatever you run there will minimally impact the switch functionality, since this environment and all the software running on it will be running in the user space of the underlying Linux system—and not in the kernel space.

Figure 17-1 illustrates the software architecture of a Nexus switch running NX-OS.

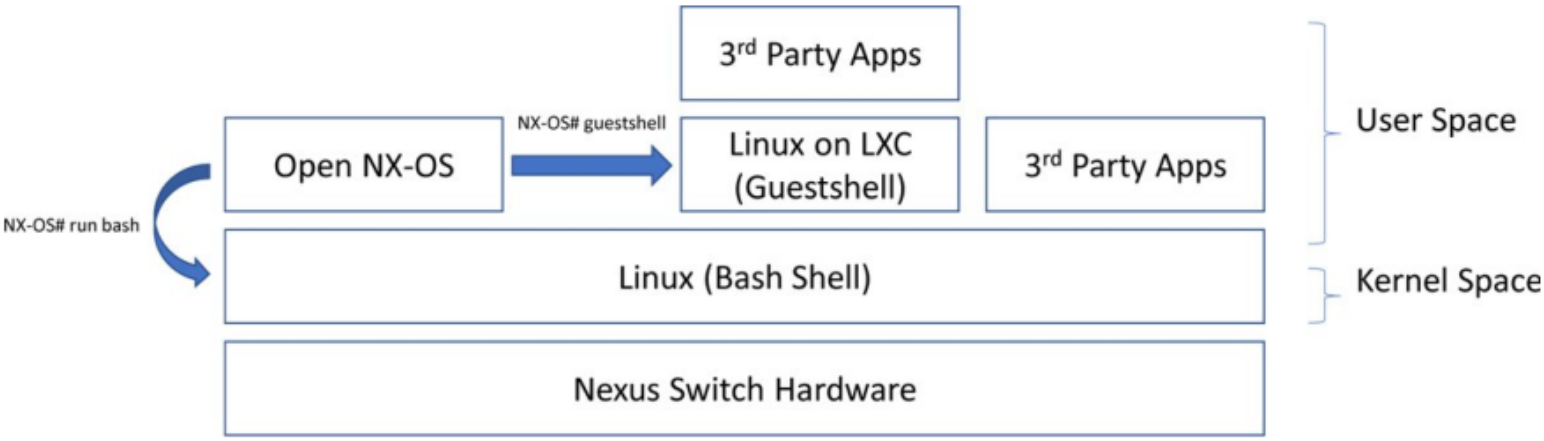

Figure 17-1 *NX-OS Software Architecture*

Before you can access the Bash shell, you need to enable it by using the command **feature bash** and then start the shell by using the command **run bash**, as shown in Example 17-1.

Example 17-1 *Enabling the Bash Shell and Accessing It on NX-OS*

```
sbx-n9kv-ao# config terminal

Enter configuration commands, one per line. End with CNTL/Z.
```

```
sbx-n9kv-ao(config)# feature bash-shell

sbx-n9kv-ao(config)# show feature | in bash

bash-shell            1          enabled

sbx-n9kv-ao(config)# exit

sbx-n9kv-ao# run bash

bash-4.3$

bash-4.3$ exit

exit

sbx-n9kv-ao#
```

Example 17-2 shows the output of the **uname -a** command as well as the contents of the /etc/os-release file. As you can see, the switch is running on a Wind River Linux distro (cisco-wrlinux) Version 8, based on, as of this writing, Linux kernel Version 4.1.

Example 17-2 *Bash Shell Details on NX-OS*

```
bash-4.3$ uname -a

Linux sbx-n9kv-ao 4.1.21-WR8.0.0.25-standard #1 SMP Tue Jul 17 15:34:01 PDT 2018 x86_64

x86_64 x86_64 GNU/Linux

bash-4.3$ cat /etc/os-release

ID=nexus

ID_LIKE=cisco-wrlinux

NAME=Nexus

VERSION="9.2(1)I9(1)"

VERSION_ID="9.2(1)I9"

PRETTY_NAME="Nexus 9.2(1)I9"

HOME_URL=http://www.cisco.com

BUILD_ID="9.2(1)I9(1)"

CISCO_RELEASE_INFO=/etc/os-release

bash-4.3$ exit

exit

sbx-n9kv-ao#
```

A few things are immediately noticeable when you run the Bash shell: The familiar [*user@hostname* ~]$ prompt that you saw in Chapter 2 is not present unless you switch to the root user. And if the directory is changed using the **cd** command, the prompt does not change to reflect the current directory. Instead, the **id** and **whoami** commands can be used to identify the current user, and the **pwd** command can be used to identify the current working directory, as shown in Example 17-3.

Example 17-3 *The **whoami**, **id**, and **pwd** Commands in the Bash Shell*

```
bash-4.3$ whoami

admin

bash-4.3$ id

uid=2002(admin) gid=503(network-admin) groups=503(network-admin),504(network-operator)

bash-4.3$ pwd

/bootflash/home/admin

! Switching to the root user

bash-4.3$ su -

Password:

root@sbx-n9kv-ao# whoami

root

root@sbx-n9kv-ao# id

uid=0(root) gid=0(root) groups=0(root)

root@sbx-n9kv-ao# pwd

/root

root@sbx-n9kv-ao#
```

Notice, however, that the prompt ending changes from the $ sign to the # sign when you switch to the root user, as expected.

Getting help in the Nexus Bash shell can be tricky. Despite the fact that you frequently get a message asking you to use **man** {*command*} or **info** {*command*} to check the man or info pages for a command, the man and info pages may not actually exist on some platforms. In such cases, you can use {*command*} --**help** to get some help on the use of a command.

Apart from these caveats, the Bash shell should feel familiar to you. The vast majority of the directory and file management commands covered in Chapters 2 through 4 should work just fine in the Bash shell. You should experiment with the Bash shell by creating, moving, copying, and deleting files and directories to see what works and what doesn't. Keep in mind, however, that deleting some files or directories may leave you

with a nonfunctioning switch since everything else depends on the underlying Linux system to work properly.

You can manage switch components, such as the switch interfaces, from the Bash shell by using the same tools and commands used to manage a regular Linux server. Example 17-4 shows the output of the **show interface brief** command at the Nexus switch CLI.

Example 17-4 *Output of the **show interface brief** Command at the CLI*

```
sbx-n9kv-ao# show interface brief


--------------------------------------------------------------------------------
Port   VRF          Status IP Address                              Speed    MTU
--------------------------------------------------------------------------------
mgmt0  --           up     10.10.20.95                             1000     1500


--------------------------------------------------------------------------------
Ethernet      VLAN    Type Mode   Status  Reason                   Speed     Port
Interface                                                                    Ch #
--------------------------------------------------------------------------------
Eth1/1        1       eth  trunk  up      none                       1000(D) 11
Eth1/2        1       eth  trunk  up      none                       1000(D) 11


------ output omitted for brevity ------


Eth1/127      1       eth  access down    Link not connected         auto(D) --
Eth1/128      1       eth  access down    Link not connected         auto(D) --


------ output omitted for brevity ------


sbx-n9kv-ao#
```

Example 17-5 shows corresponding output from the Bash shell. In Chapter 3 you learned that the commands **ip link** and **ip address** display the interfaces status on a server. Example 17-5 shows the output of the first of these commands when entered in the Bash shell on the switch.

Example 17-5 *Output of the **ip link** Commands in the Bash Shell*

```
bash-4.3$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: dummy0: <BROADCAST,NOARP> mtu 1500 qdisc noop state DOWN mode DEFAULT group default
    link/ether fa:aa:16:00:0d:16 brd ff:ff:ff:ff:ff:ff


------ output omitted for brevity ------


171: Eth1-127: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 100
    link/ether 00:50:56:bb:ab:85 brd ff:ff:ff:ff:ff:ff
172: Eth1-128: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 100
    link/ether 00:50:56:bb:ab:86 brd ff:ff:ff:ff:ff:ff
bash-4.3$ #
```

In a similar fashion, package and repository management on a Nexus switch can be managed from the Bash shell by using the familiar **rpm** and/or **yum** tools.

A useful functionality available in the Bash shell is the capability to run NX-OS CLI commands. This comes in handy if you wish to run a Bash script that directly invokes a CLI command or simply use the Linux tools **grep**, **awk**, and **sed** (covered in Chapter 4) to filter or edit command output. Example 17-6 shows the **vsh** command being used to extract all /32 routes in the routing table, **grep** being used to match on the subnet mask, and the output being piped to **awk** to print the subnet only (without the trailing comma). This example gives you an idea of how useful it would be to be able to manipulate command output with the full arsenal of Bash commands and utilities available to you through the Bash shell.

Example 17-6 *Using Bash Utilities to Filter and Edit Command Output from the Bash Shell*

```
bash-4.3$ vsh -c "show ip route" | grep '/32' | awk -F , '{print $1}'
172.16.0.1/32
172.16.1.1/32
172.16.100.1/32
172.16.101.1/32
```

```
172.16.102.1/32

172.16.103.1/32

172.16.104.1/32

172.16.105.1/32

bash-4.3$ #
```

For a review of the **grep, sed** and **awk** utilities, see Chapter 4.

As mentioned earlier, the Guest shell provides access to a Linux environment that runs in an LXC container on top of the underlying Linux OS. The Guest shell provides access to the following:

- The switch interfaces through the underlying Linux OS network subsystem for monitoring only
- The switch file system
- The switch RAM (volatile tmpfs)
- The NX-OS CLI
- The NX-API REST API

You invoke the Guest shell by using the command **guestshell** at the NX-OS CLI, which initiates an SSH connection to the Guest shell Linux system. Unlike the Bash shell, the Guest shell is actually a CentOS Version 7 distro, as you can see in Example 17-7.

Example 17-7 *Invoking the Guest Shell from the CLI*

```
sbx-n9kv-ao# guestshell

[admin@guestshell ~]$ uname -a

Linux guestshell 4.1.21-WR8.0.0.25-standard #1 SMP Tue Jul 17 15:34:01 PDT 2018 x86_64

x86_64 x86_64 GNU/Linux

[admin@guestshell ~]$ cat /etc/os-release

NAME="CentOS Linux"

VERSION="7 (Core)"

ID="centos"

ID_LIKE="rhel fedora"

VERSION_ID="7"

PRETTY_NAME="CentOS Linux 7 (Core)"

ANSI_COLOR="0;31"

CPE_NAME="cpe:/o:centos:centos:7"

HOME_URL="https://www.centos.org/"

BUG_REPORT_URL="https://bugs.centos.org/"


CENTOS_MANTISBT_PROJECT="CentOS-7"

CENTOS_MANTISBT_PROJECT_VERSION="7"

REDHAT_SUPPORT_PRODUCT="centos"

REDHAT_SUPPORT_PRODUCT_VERSION="7"


CISCO_RELEASE_INFO=/etc/shared/os-release

[admin@guestshell ~]$
```

In addition to offering limited access, the Guest shell runs on capped resources. This means that whatever runs in the Guest shell will not exceed certain limits with respect to CPU, memory, and storage, and it will therefore never adversely affect the basic functionality of the underlying Linux system or NX-OS. These limits are, however, configurable.

Both IOS XE and IOS XR provide access to Linux shells. However, IOS XR only provides a Bash shell to the underlying Linux OS on which IOS XR is running, and you can access it by running the CLI command **bash**. On the other hand, IOS XE only exposes a Guest shell that runs in a container and is isolated from the underlying Linux OS; you can access it by using the command **guestshell**.

Use Case 2: NX-API CLI

The NX-API CLI is an API exposed by NX-OS that provides the facility to encode (encapsulate) CLI or Linux Bash commands in JSON or XML and send them to the switch over HTTPS. The message body of the HTTP response, which is the command output in this case, is also received in JSON or XML. NX-API uses HTTP Basic Authentication.

In order to start using the NX-API CLI, you need to enable the feature by issuing the command **feature nxapi**. You can confirm that it is enabled by using the command **show feature | incl nxapi**. The feature should show up as **enabled**. You can also use the command **show nxapi** to check the status of the NX-API and its configuration.

You can use the NX-API CLI off box via any HTTP client, such as Postman or Python HTTP libraries, and you can also use it on box via a tool provided by Cisco called the *Developer Sandbox*.

Note

Don't confuse this with the sandboxes provided by Cisco DevNet for testing purposes. Whereas the Developer Sandbox is a feature, the DevNet sandboxes are test devices provided for testing purposes in isolation from any production network.

To access the Developer Sandbox, you use a web browser to browse to the switch name or IP address. For the DevNet always-on NX-OS sandbox, open your browser and browse to the URI https://sbx-nxos-mgmt.cisco.com. When prompted for a username and password, enter **admin** and **Admin_1234!**. You are then presented with the GUI shown in Figure 17-2. As of this writing, this particular switch runs NX-OS Version 9.3(3).

Note

The developer sandbox GUI may change from what is shown in Figure 17-2 with new NX-OS versions. Visit the DevNet sandboxes web page (https://devnetsandbox.cisco.com/RM/Topology) to get the updated URI and credentials for the NX-OS DevNet sandbox.

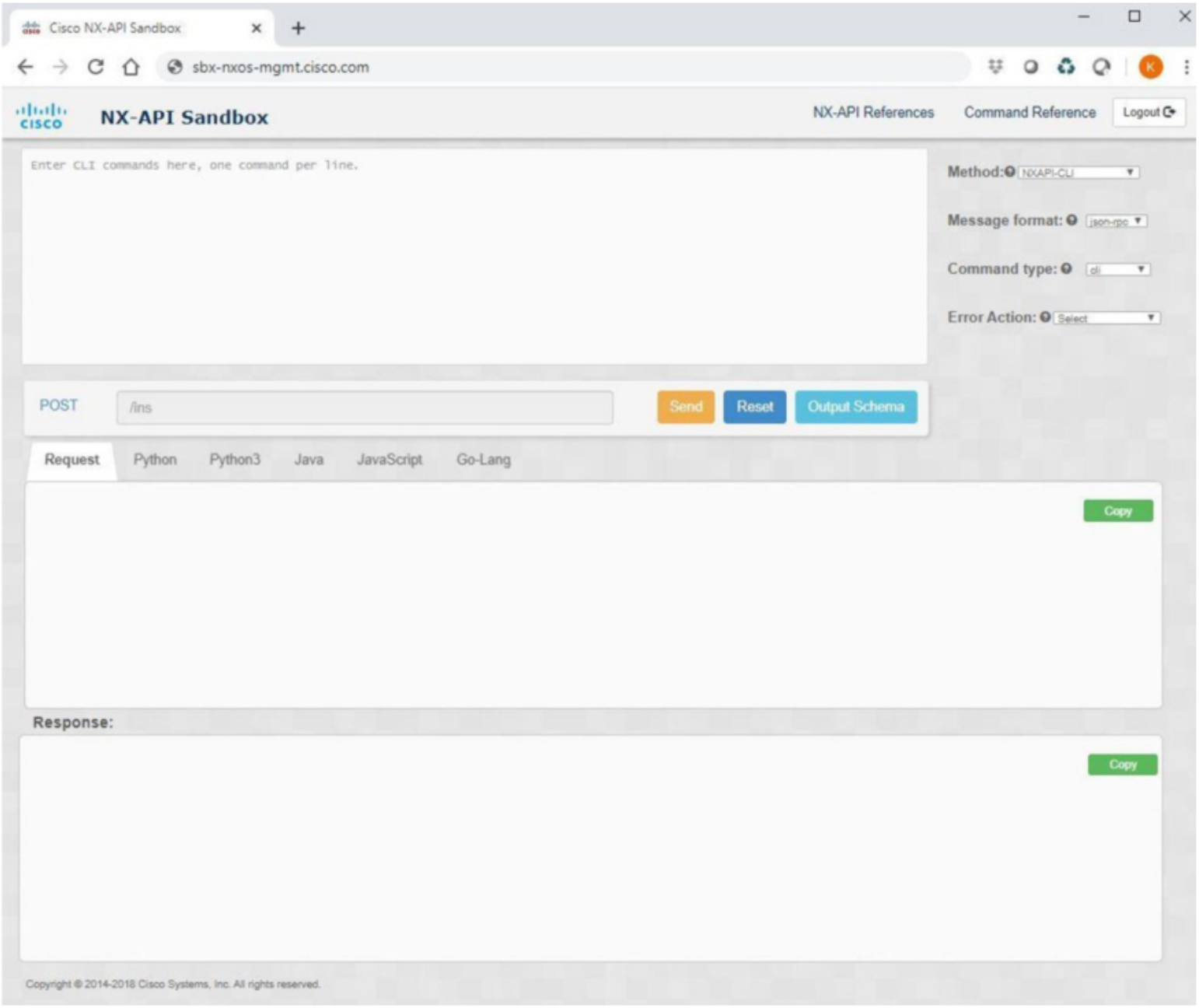

Figure 17-2 *The Developer Sandbox Graphical User Interface*

In order to understand how the developer sandbox assists you with converting CLI commands to API calls, let's examine the sandbox GUI. The GUI has three panes:

• The top pane is the Command pane, and this is where your input goes.

• The middle pane is the Request pane. The API payload body equivalent to the command that you enter in the Command pane appears in the Request pane.

• The pane at the bottom is the Response pane. This is where the response from the switch shows up when you click the Send button.

Caution

As you proceed, keep in mind that any command entered in the developer sandbox and sent to the switch will actually be executed. Therefore, you should take extra care before clicking Send when using the sandbox on a network element in a production network.

To the right of the Command pane are four drop-down boxes: Method, Message Format, Command Type, and Error Action.

The Method drop-down enables you to select which API or protocol to work with. The three options here are NXAPI-CLI, NXAPI-REST (DME), and RESTCONF (Yang). (This section covers the NXAPI-CLI option, and the next section covers the NXAPI-REST (DME) option.)

When you choose NXAPI-CLI from the Method drop-down, you get three options in the Message format drop-down: json-rpc, xml, and json. Depending on which one you choose, you will be presented with up to five options for the Command Type drop-down (as discussed later in this section).

You can try out this simple example to get started: Enter the command **show hostname** in the Command pane and choose json from the Message format drop-down and cli_show from the Command type drop-down. As soon as you enter the command in the Command pane, the Request pane is updated with a JSON object. This is the *payload body* of the API call to the switch. The full API details are as follows:

• **Method**: POST

• **URI**: https://sbx-nxos-mgmt.cisco.com/ins/

• **Header**:

```
Content-Type: application/json
```

• **JSON payload**:

```
{
  "ins_api": {
    "version": "1.0",
    "type": "cli_show",
    "chunk": "0",
    "sid": "1",
    "input": "show hostname",
    "output_format": "json"
  }
}
```

• **Authorization**: One of the two following options:

• Choose Basic Auth and provide username/password (**admin**/**Admin_1234!** for this example).

• Authenticate once via a POST request by providing username/password, getting a cookie, and then adding the **Cookie** header in all subsequent requests.

API calls to the NX-API CLI always use the POST method and the URI **https://**{*switch_address*}**/ins/**. The header Content-Type depends on the payload *encoding*, which in turn depends on the Message format setting you choose. The payload body *content* depends on the protocol (or API) you are going to use, which again is based on your selection from the Message format drop-down. Finally, authentication is always required unless you are doing the API call on box from the developer sandbox by clicking the Send button under the Command pane.

Now click the Send button and watch the switch response, encoded as a JSON object, appear in the Response pane. The switch hostname in this case is sbx-n9kv-ao.

To elaborate more on what exactly has happened here, you can make the same API call from Postman. Open Postman and create a new request. Edit the request attributes to match the same method, URI, headers, payload, and authentication settings shown earlier. You should then send the request and notice the body of the HTTP response. Figure 17-3 shows a screenshot of the developer sandbox GUI and the Postman application after you make the API call from both of them. Compare the response bodies in the figure, and you see that they are identical.

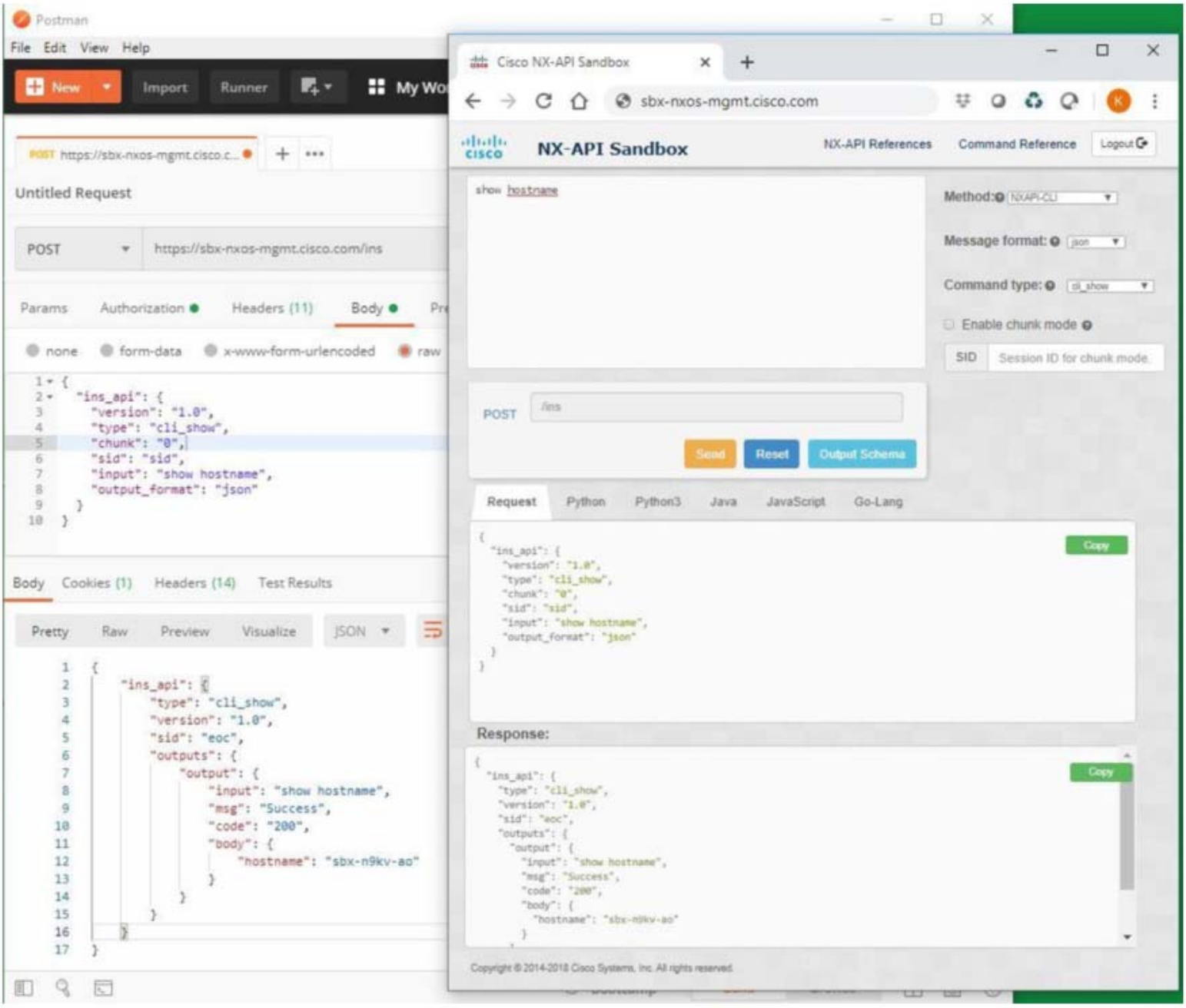

Figure 17-3 *Comparing the Body of the Response in the Developer Sandbox and Postman*

To use this API, you must provide the switch with a username and password for basic HTTP authentication. Alternatively, you might want to save a few of the processing cycles involved with authenticating each and every request by authenticating once and receiving a cookie from the server in the form of a response header field called Set-Cookie. The cookie is stored on your client and should be included in all subsequent HTTP requests to that switch, as the value of the request header field Cookie.

To authenticate and receive a cookie, send a POST request to the URI https://sbx-nxos-mgmt.cisco.com/api/aaaLogin.json with the request body content highlighted in Example 17-8. Notice that the *response* in the example has a header field called Set-Cookie. The value of this header field should be included in all subsequent HTTP *requests* as the value of the header field Cookie. The cookie value is automatically included for you in Postman as soon as you send a request to the same host from which you received the cookie. The session cookie expires after 10 minutes, and this amount of time is not configurable.

Example 17-8 *Authenticating Once to Receive a Cookie in the Set-Cookie Response Header Field*

```
! HTTP Request

POST /api/aaaLogin.json HTTP/1.1

Content-Type: application/json

Cache-Control: no-cache

User-Agent: PostmanRuntime/7.24.0

Accept: */*

Host: sbx-nxos-mgmt.cisco.com

Accept-Encoding: gzip, deflate, br

Connection: keep-alive

Content-Length: 116

{

     "aaaUser": {

       "attributes": {
```

```
          "name": "admin",
          "pwd": "Admin_1234!"
        }
    }
}


! HTTP Response
HTTP/1.1 200 OK
Server: nginx/1.13.12
Date: Wed, 01 Apr 2020 23:39:26 GMT
Content-Type: application/json
Content-Length: 891
Connection: keep-alive
Set-Cookie: APIC-
cookie=DIq05JJ4w5ElHP2xG3Z5+R0eGWTcT2ZM5QDfA2OAvlx7DGY0kj70198sbnWy73wcOAAjELeV9nrVarLq/
r9Dp8vsEjoTelhEVXI590CBjruoniT6+flZI3olS7411OdDrNRjkz3DDYABLiNdGNXxqKdkNJ7WxNFQrVCc+NfmX
sQ=; path=/; HttpOnly
------ output truncated for brevity ------
```

The way the Developer Sandbox works and how it can help you when writing automation code should be clear to you by now. One very handy feature of the application is that it can also generate Python 2 and 3, Java, JavaScript, or Go code that needs minor editing to be ready for use. You can click the corresponding tab button in the Request pane (refer to Figures 17-2 and 17-3) to get the code in the language that you need. That code can then be used off box to perform the same API call from a machine that can reach this switch.

When the Method drop-down in the Command pane is set to NXAPI-CLI, the Message Format drop-down list provides three different formats. The json-rpc option generates a payload that conforms with the standards-based lightweight protocol JSON-RPC 2.0 (defined at https://www.jsonrpc.org/specification). The xml and json options generate a payload in XML or JSON, respectively, to be used by proprietary Cisco protocols.

In the Command type drop-down, you specify whether the command that you provide in the Command pane is a CLI **show** command, a CLI configuration command, or a Bash shell command.

When in doubt about any of the options, you can hover with your mouse pointer over the question mark to the left of a drop-down list to get a brief description of the option.

Use Case 3: NX-API REST

NX-API REST is a RESTful API exposed by NX-OS. Like any other RESTful API, the API is defined through a set of URIs (endpoints) and the parameters in those URIs, the methods allowed on the resources represented by those URIs, header fields, the encoding of a message body, and the content of that body, as determined by the data model used by the API. (The NX-API REST API is used in the examples in Chapter 7 to demonstrate the different components of HTTP.)

The NX-API REST API uses the GET, POST, PUT, DELETE, and OPTIONS methods. It supports payload bodies encoded in either XML or JSON.

REST APIs act on resources or, more accurately, representations of resources. In Cisco lingo, these resources are called *objects*, and instances of these objects are referred to as *managed objects* (*MO*s). MOs represent configuration and state data and are organized in a hierarchical tree structure called the *management information tree* (*MIT*).

Each MO represents a resource on the switch that can be individually managed using the API. An MO is uniquely and globally identified by a *distinguished name* (*DN*). The DN is the full path to the MO and describes its location in the MIT, starting at the root of the tree; the DN therefore contains all of the MO's parent objects. As you might have guessed, the DN value constitutes most of the URI used to address the MO. The individual MO's name, excluding the hierarchical part of the DN, is called the *relative name* (*RN*). For example, if the DN of interface Eth1/1 is sys/intf/phys-[eth1/1], the RN is phys-[eth1/1]. Keep in mind that phys-[eth1/1] is an instance of the object named l1PhysIf. Also keep in mind that each node of the MIT (not only the leafs) constitutes an MO. So sys is an MO, and intf is an MO as well. You need to know this because constructing a URI involves knowing the DN of the MO you are trying to address, and constructing the body of the message involves understanding the MIT hierarchy and attributes of each MO.

The general URI format for addressing an MO is **https://**{*SW-FQDN*|*IP*}**/api/mo/**{*DN*}.{**json**|**xml**}[**?***options*]. MOs have attributes. For example, an interface may be a layer 2 or layer 3 interface. If it is a layer 2 interface, it has its trunking mode set to either access or trunk. If it is an access port, it has a VLAN assigned, and if it is a trunk port, it has a set of allowed VLANs. The interface is the MO, and the layer, trunking mode, and VLAN numbers are some of the attributes of that MO. Some attributes have default values, and some do not.

Apart from having values, attributes have types. An MTU value is an integer. It cannot contain decimal places, and it is not a string, so it cannot contain letters. The admin state is Boolean, and it can be either up or down. A switch has an internal engine called the *Data Management Engine* (*DME*) that maintains all the attribute names and properties for all MOs, including their default values, and validates any value that you attempt to assign to these attributes.

The MIT and the DME together constitute the data model referenced by this API. (For more information on what a data model is, refer to Chapter 13, "YANG.") This means that the content of the HTTP request or response body is primarily determined by the attributes you are attempting to configure (in the case of a POST). HTTP requests used to retrieve MO information by using **GET** do not have a message body. However, the HTTP response message body follows the same data model.

How do you figure out the DN of an MO in order to build the URI? There are a few ways to do this. First, there is an on-box feature called Visore. To access Visore on the NX-OS sandbox, go to https://sbx-nxos-mgmt.cisco.com/visore.html and log in using the same credentials you use for the Developer Sandbox. In the Class or DN text box at the top of the page, type **sys** and click the Run Query button. The sys MO details then appear, as shown in Figure 17-4.

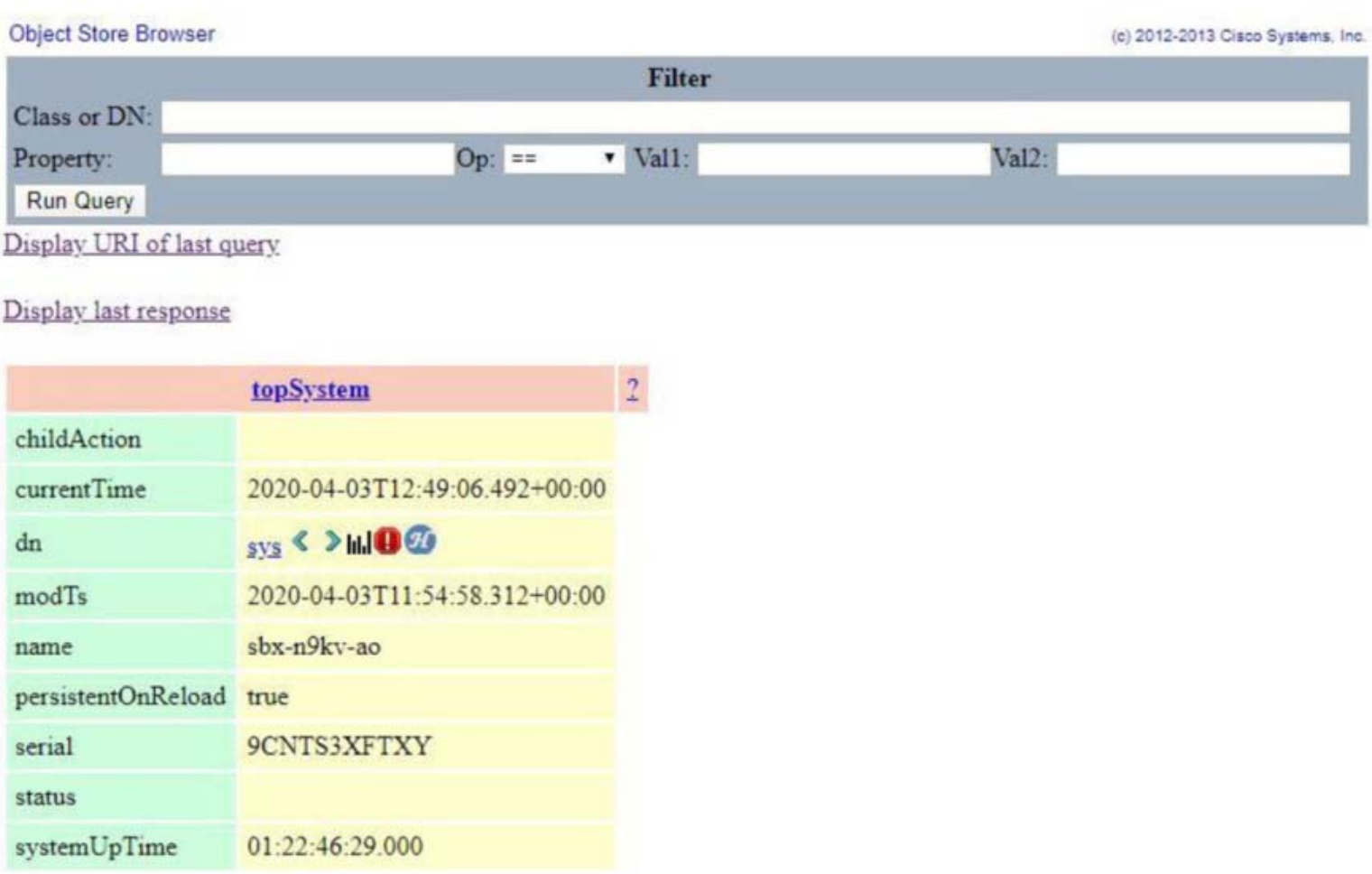

Figure 17-4 *Details of the sys MO in Visore*

Click the small green arrow to the right of sys in the dn field. You then see all the child objects under the topSystem object sys. Although it is not very efficient, you can scroll or search (by pressing Ctrl+F) through the results to find the interfaceEntity object with the RN intf. Then you can click the green arrow to the right of the DN sys/intf, and you are presented with a list of l1PhysIf objects, one for each physical interface. Again, scroll or search for your interface until you find the needed DN. (Keep in mind that Visore, like all other features and tools provided by Cisco, may at any point in time be deprecated or have its GUI changed. The description given here is valid as of this writing.)

Another way to figure out the DN of an MO in order to build the URI is to consult the Cisco Nexus 3000 and 9000 Series NX-API REST documentation at https://developer.cisco.com/docs/cisco-nexus-3000-and-9000-series-nx-api-rest-sdk-user-guide-and-api-reference-release-9-3x/. This documentation provides step-by-step instructions on how to query or configure any MO.

Now that you know how to figure out the DN of the MO, let's move to some hands-on activities.

NX-API CLI and REST use the same set of commands to enable the APIs and check their status. Use the command **feature nxapi** to enable the feature and **show nxapi** to check the status of the API, as well as which HTTP/HTTPS port the switch is listening on. The **show nxapi** command also displays the switch certificate information.

Let's go back to the earlier example of the layer 2 interface. Say that you need to configure interface Eth1/5 on the Cisco NX-OS sandbox as a trunk port allowing VLANs 1 to 100. Before configuring the interface, you should retrieve its current configuration. Referring to the API reference or Visore, you can send a **GET** request to the URI https://sbx-nxos-mgmt.cisco.com/api/mo/sys/intf/phys-[eth1/5].json. Example 17-9 shows the HTTP response body.

Example 17-9 *Retrieving the Interface Information for Interface Eth1/5*


```
{
    "totalCount": "1",
    "imdata": [
        {
            "l1PhysIf": {
                "attributes": {
                    "FECMode": "auto",
                    "accessVlan": "vlan-1",
                    "adminSt": "up",
                    "autoNeg": "on",
                    "beacon": "off",
                    "bw": "default",
                    "childAction": "",
                    "controllerId": "",
                    "delay": "1",
                    "descr": "",
```

```
                    "dn": "sys/intf/phys-[eth1/5]",
                    "dotlqEtherType": "0x8100",
                    "duplex": "auto",
                    "ethpmCfgFailedBmp": "",
                    "ethpmCfgFailedTs": "00:00:00:00.000",
                    "ethpmCfgState": "0",
                    "id": "eth1/5",
                    "inhBw": "4294967295",
                    "layer": "Layer2",
                    "linkDebounce": "100",
                    "linkDebounceLinkUp": "0",
                    "linkLog": "default",
                    "linkTransmitReset": "enable",
                    "mdix": "auto",
                    "medium": "broadcast",
                    "modTs": "2020-04-03T00:43:46.988+00:00",
                    "mode": "access",
                    "mtu": "1500",
                    "name": "",
                    "nativeVlan": "vlan-1",
                    "persistentOnReload": "true",
                    "portT": "leaf",
                    "routerMac": "not-applicable",
                    "snmpTrapSt": "enable",
                    "spanMode": "not-a-span-dest",
                    "speed": "auto",
                    "speedGroup": "auto",
                    "status": "",
                    "switchingSt": "disabled",
                    "trunkLog": "default",
                    "trunkVlans": "1-4094",
                    "usage": "discovery",
                    "userCfgdFlags": "",
                    "vlanmgrCfgFailedBmp": "",
                    "vlanmgrCfgFailedTs": "00:00:00:00.000",
                    "vlanmgrCfgState": "0",
                    "voicePortCos": "none",
                    "voicePortTrust": "disable",
                    "voiceVlanId": "none",
                    "voiceVlanType": "none"
                }
            }
        }
    ]
}
```


A few fields in this output are highlighted. You can see that the interface is a layer 2 interface, in access mode, and the access VLAN is VLAN 1, which is the default interface configuration on the Nexus switch. Notice also that this MO is an instance of an l1PhysIf object, and its DN is sys/intf/phys-[eth1/5].

Referring to the references mentioned earlier, you can configure the interface as stated by executing an API call with the following details:

• **Method**: POST

• **URI**: http://sbx-nxos-mgmt.cisco.com/api/mo/sys/intf.json

• **JSON payload:**


```
{
```

```
  "interfaceEntity": {
    "children": [
      {
        "l1PhysIf": {
          "attributes": {
            "id": "eth1/5",
            "mode": "trunk",
            "trunkVlans": "1-100"
          }
        }
      }
    ]
  }
}
```

• **Authorization**: Authenticate once via a POST request by providing the username/password, getting a cookie, and then adding the Cookie header in all subsequent requests. NX-API REST does not support direct HTTP Basic Authentication.

Before proceeding, it is worth noting that one of the ways to figure out the payload for the API call is through the Developer Sandbox. Go to the Sandbox GUI and choose the NXAPI-REST (DME) from the Method drop-down and cli from the Input type drop-down. Enter the configuration commands as you would on the switch CLI, one command per line, and then click Convert. The required payload then appears in the Request pane (see Figure 17-5).

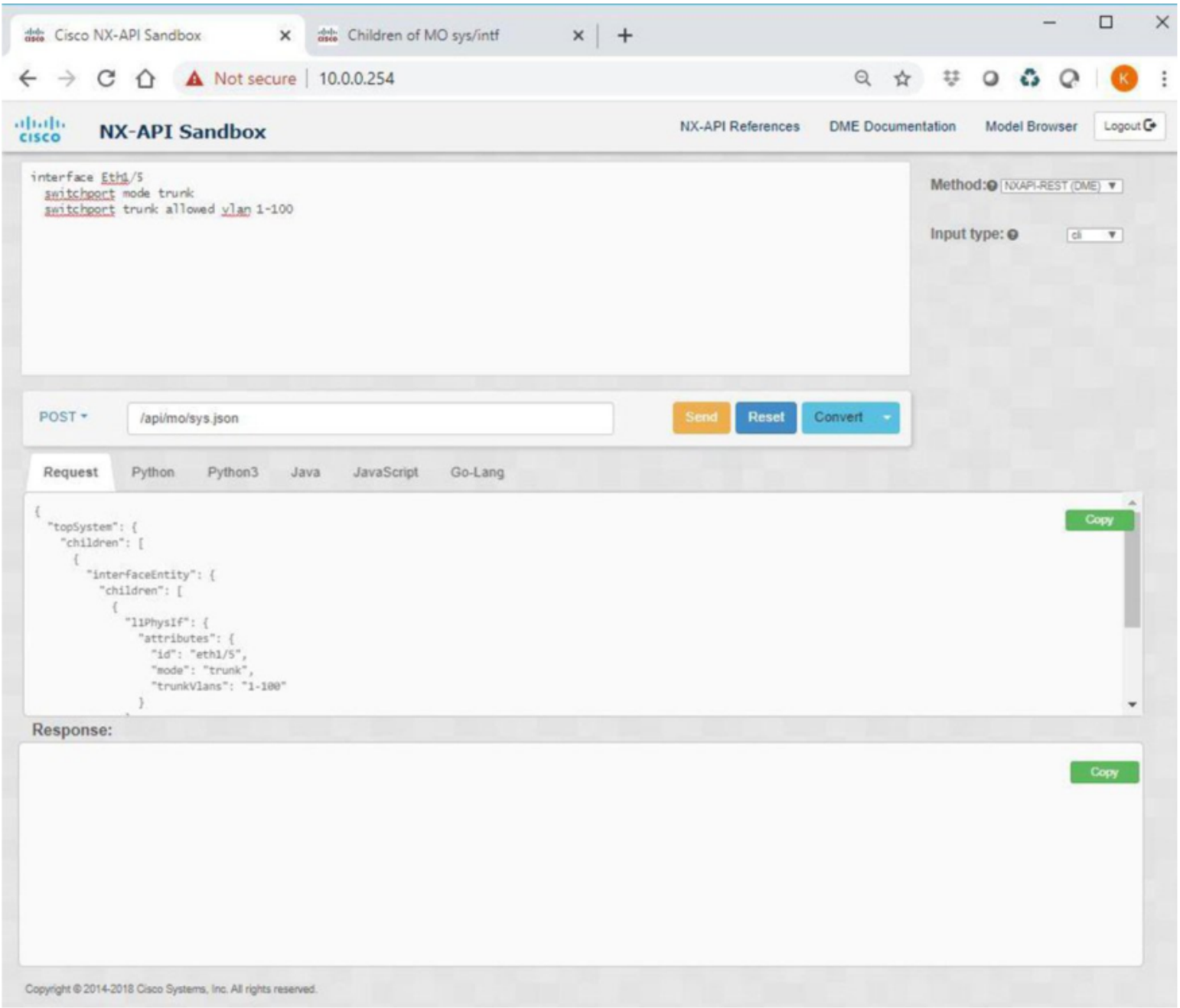

Figure 17-5 *Using the Developer Sandbox to Translate CLI Configuration Commands to an NX-API REST Payload*

Compare what you see in the figure with what's in the bulleted list, copied from the Cisco documentation. Note that the DN + body content combination in the bulleted list is slightly different from the DN + body content from the Developer Sandbox. Both communicate the same

information through an API call. The difference is that the API call in the bulleted list provides a little more hierarchy information through the URI, and the API call from the developer sandbox uses a single URI (**https://sbx-nxos-mgmt.cisco.com/api/mo/sys.{json|xml}**) and provides all other necessary information through the message body content.

Now open Postman and construct a new request, using the information in the bulleted list, and send the request. You should expect a 200 OK response. This POST request addresses the MO named interfaceEntity, which is the parent of the MO named l1PhysIf. The DN of this parent MO is sys/intf. There is no real need to memorize either the DN or the content structure. (Refer to the Cisco documentation, if needed.)

Now you can send another GET request to retrieve the interface attributes. Example 17-10 shows output confirming that all went as intended.

Example 17-10 *Retrieving the Interface Information for Interface Eth1/5 After the Configuration*


```
{
    "totalCount": "1",
    "imdata": [
        {
            "l1PhysIf": {
                "attributes": {
                    "FECMode": "auto",
                    "accessVlan": "vlan-1",
                    "adminSt": "up",
                    "autoNeg": "on",
                    "beacon": "off",
                    "bw": "default",
                    "childAction": "",
                    "controllerId": "",
                    "delay": "1",
                    "descr": "Link to virt3",
                    "dn": "sys/intf/phys-[eth1/5]",
                    "dotlqEtherType": "0x8100",
                    "duplex": "auto",
                    "ethpmCfgFailedBmp": "",
                    "ethpmCfgFailedTs": "00:00:00:00.000",
                    "ethpmCfgState": "0",
                    "id": "eth1/5",
                    "inhBw": "4294967295",
                    "layer": "Layer2",
                    "linkDebounce": "100",
                    "linkDebounceLinkUp": "0",
                    "linkLog": "default",
                    "linkTransmitReset": "enable",
                    "mdix": "auto",
                    "medium": "broadcast",
                    "modTs": "2020-04-03T13:26:22.570+00:00",
                    "mode": "trunk",
                    "mtu": "1500",
                    "name": "",
                    "nativeVlan": "vlan-1",
                    "persistentOnReload": "true",
                    "portT": "leaf",
                    "routerMac": "not-applicable",
                    "snmpTrapSt": "enable",
                    "spanMode": "not-a-span-dest",
                    "speed": "auto",
                    "speedGroup": "auto",
                    "status": "",
                    "switchingSt": "disabled",
                    "trunkLog": "default",
                    "trunkVlans": "1-100",
```

```
                    "usage": "discovery",
                    "userCfgdFlags": "",
                    "vlanmgrCfgFailedBmp": "",
                    "vlanmgrCfgFailedTs": "00:00:00:00.000",
                    "vlanmgrCfgState": "0",
                    "voicePortCos": "none",
                    "voicePortTrust": "disable",
                    "voiceVlanId": "none",
                    "voiceVlanType": "none"
                }
            }
        }
    ]
}
```

The highlighted attributes in the example confirm that the configuration went through.

Use Case 4: NETCONF

This use case covers the NETCONF API on each of the networking operating systems: NX-OS, IOS XE, and IOS XR. The NETCONF protocol is covered in detail in Chapter 14. This section discusses the specific configuration details of NETCONF on each of the three platforms.

NETCONF on NX-OS

Nexus switches expose *two* NETCONF APIs. One API uses XSD models, and the other uses YANG models. This section covers the two NETCONF APIs on Nexus switches. The examples in this section use the NX-OS DevNet sandboxes at https://devnetsandbox.cisco.com/RM/Topology running Open NX-OS Version 9.x.

You configure the YANG-based NETCONF API, also referred to as the *NETCONF agent*, on a Nexus switch by enabling the NETCONF feature using the command **feature netconf**; you disable it by prepending **no** to the same command. You can use the command **show feature | i netconf** to make sure the feature is enabled. You can also verify the status of the NETCONF service from the Bash shell of a switch by using the command **service netconf status**. Example 17-11 shows the output of this command when the agent is running normally.

Example 17-11 *Checking the Status of the NETCONF Agent from the Bash Shell*

```
sbx-n9kv-ao# run bash
bash-4.3$ service netconf status
xosdsd (pid 30054) is running...
netconf (pid 30061) is running...
bash-4.3$
```

Two optional parameters related to NETCONF session management are **idle_timeout** and **sessions**. The first parameter, which you set by using the command **netconf idle_timeout** {*minutes*}, specifies when idle clients are disconnected, in minutes, with a default value of 5 minutes. Setting it to 0 disables timeout altogether. The second parameter, which you set by using the command **netconf sessions** {*max-sessions*}, specifies the maximum allowed number of concurrent client sessions, with a default value of 5 sessions and a maximum of 10.

To open a NETCONF session with the Nexus switch, you simply issue the command **ssh -p** {*netconf_port*} {*username*}**@**{*switch-ip-url*} **-s netconf** and, after entering the password at the password prompt, you are immediately presented with a hello message from the switch. From this point onward, everything covered in Chapter 14 is applicable here. The default port for NETCONF is 830. Example 17-12 shows the switch hello message during NETCONF session establishment.

Example 17-12 *Server Hello Message During NETCONF Session Establishment with the YANG-Based API*

```
[kabuelenain@localhost ~]$ ssh -p 830 admin@10.10.20.58 -s netconf


Welcome to the DevNet Reservable Sandbox for Open NX-OS


You can use this dedicated sandbox space for exploring and
testing APIs, explore features, and test scripts.


The following programmability features are already enabled:

  - NX-API
  - NETCONF, RESTCONF, gRPC
  - Native NX-OS and OpenConfig YANG Models


Thanks for stopping by.
Password:
<?xml version="1.0" encoding="UTF-8"?>
```

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <capabilities>
        <capability>urn:ietf:params:netconf:base:1.0</capability>
        <capability>urn:ietf:params:netconf:base:1.1</capability>
        <capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
        <capability>urn:ietf:params:netconf:capability:rollback-on-
error:1.0</capability>
        <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
        <capability>urn:ietf:params:netconf:capability:validate:1.1</capability>
        <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.1</capability>
        <capability>urn:ietf:params:netconf:capability:notification:1.0</capability>
        <capability>urn:ietf:params:netconf:capability:interleave:1.0</capability>
        <capability>urn:ietf:params:netconf:capability:with-defaults:1.0?basic-
mode=report-all</capability>
        <capability>http://cisco.com/ns/yang/cisco-nx-os-device?revision=2020-07-
20&amp;module=Cisco-NX-OS-device</capability>
    </capabilities>
    <session-id>587648866</session-id>
</hello>
]]>]]>
```

The second NETCONF API is also known as the *XML management interface*, or *XML server*, and is enabled by running the **xmlagent** service on the switch. This API is enabled by default. You can configure session timeout with the CLI command **xml server timeout** {*seconds*}, with a maximum value of 1200 seconds, which is also the default value. The maximum number of concurrent sessions can be configured by using the command **xml server max-sessions** {*sessions*}, with a maximum value of 8, which is also the default value.

You can verify the status of the XML server by using the command **show xml server status**. This command also shows all active sessions, as shown in Example 17-13.

Example 17-13 *Checking the Status of the XML Server from the CLI*

```
sbx-n9kv# show xml server status
 operational status is enabled
 maximum session configured is 8
 session: 25118, user: admin, starttime: Thu Nov 26 21:50:54 2020, sap: 9648 timeout:
1200, time remaining: 1200 ip address: 192.168.254.11
```

Notice the highlighted session ID value in the example. You can use this value in the command **xml server terminate** {*session-id*} to terminate that active session.

NETCONF supports the concept of multiple datastores. Open NX-OS supports running configuration and candidate configuration datastores. You can apply configuration changes directly to the running configuration or choose to apply the changes to the candidate configuration first, validate the candidate configuration, and then commit it to the running configuration.

As you have read in Chapter 14, the datastore to apply the changes to is called the *target*, and it is specified in the NETCONF message from the client. You may also specify a *source*. This is required in some cases, such as when the candidate configuration is being initialized for the first time. In that case, the target is the candidate configuration, and the source is the running configuration. When initializing the candidate configuration for the first time, the source cannot be anything except the running configuration.

To access the XML server API, you can use either of these methods:

• Use SSH to the xmlagent subsystem on the switch by entering the command **ssh** [**-p** {*ssh_port*}] {*username*}**@**{*switch-ip-url*} **-s xmlagent**.

• Enter the CLI command **xmlagent** at the exec prompt of the switch.

Example 17-14 shows the NETCONF hello message from a Nexus switch running Open NX-OS Version 9.3(5) when using the XML-based API. Notice that the option **-p** was not used because the default SSH port is being used (port 22).

Example 17-14 *The Switch Hello Message Right After Initiating a NETCONF Session*

```
[kabuelenain@localhost ~]$ ssh admin@10.10.20.58 -s xmlagent


Welcome to the DevNet Reservable Sandbox for Open NX-OS


You can use this dedicated sandbox space for exploring and
testing APIs, explore features, and test scripts.


The following programmability features are already enabled:
  - NX-API
```

```
  - NETCONF, RESTCONF, gRPC
  - Native NX-OS and OpenConfig YANG Models


Thanks for stopping by.
Password:
<?xml version="1.0" encoding="ISO-8859-1"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:url:1.0?scheme=file</capability>
    <capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.0</capability>
  </capabilities>
  <session-id>25118</session-id>
</hello>
]]>]]>
```

As you can see, in addition to the base capabilities, the switch supports six extended capabilities.

The XSD schemas supported by the switch are located in the directory /isan/etc/schema on the switch. You can access the directory and open any of the schemas through the switch Bash shell.

Open NX-OS provides several built-in tools to assist you with constructing XML-encoded NETCONF XSD-based API messages, and it provides command output encoded in XML. While these commands may not be very helpful for the YANG-based API, they are very handy when working with the XSD-based API. These are the commands:

• **xmlin**: This is an interactive-mode tool that convert exec and config commands to their XML equivalents.

• **show** {*command*} **| xmlin**: This command provides the XML-encoded NETCONF rpc message equivalent of the **show** command.

• **show** {*command*} **| xmlout**: This command provides the XML-encoded switch output of the **show** command.

Example 17-15 shows the **xmlin** interactive tool in action.

Example 17-15 *The **xmlin** Interactive Tool*

```
sbx-n9kv-ao# xmlin
****************************************
Loading the xmlin tool. Please be patient.
****************************************
Cisco NX-OS Software
Copyright (c) 2002-2018, Cisco Systems, Inc. All rights reserved.
Nexus 9000v software ("Nexus 9000v Software") and related documentation,
files or other reference materials ("Documentation") are
the proprietary property and confidential information of Cisco
Systems, Inc. ("Cisco") and are protected, without limitation,
pursuant to United States and International copyright and trademark
laws in the applicable jurisdiction which provide civil and criminal
penalties for copying or distribution without Cisco's authorization.

Any use or disclosure, in whole or in part, of the Nexus 9000v Software
or Documentation to any third party for any purposes is expressly
prohibited except as otherwise authorized by Cisco in writing.
The copyrights to certain works contained herein are owned by other
third parties and are used and distributed under license. Some parts
of this software may be covered under the GNU Public License or the
GNU Lesser General Public License. A copy of each such license is
available at
http://www.gnu.org/licenses/gpl.html and
```

```
http://www.gnu.org/licenses/lgpl.html
*******************************************************************************
*  Nexus 9000v is strictly limited to use for evaluation, demonstration   *
*  and NX-OS education. Any use or disclosure, in whole or in part of     *
*  the Nexus 9000v Software or Documentation to any third party for any   *
*  purposes is expressly prohibited except as otherwise authorized by     *
*  Cisco in writing.                                                      *
*******************************************************************************
sbx-n9kv-ao(xmlin)# config t
Enter configuration commands, one per line. End with CNTL/Z.
sbx-n9kv-ao(config)(xmlin)# interface Eth1/10
% Success
sbx-n9kv-ao(config-if-verify)(xmlin)# switchport mode access
% Success
sbx-n9kv-ao(config-if-verify)(xmlin)# switchport access vlan 100
% Success
sbx-n9kv-ao(config-if-verify)(xmlin)# end
<?xml version="1.0"?>
<nf:rpc xmlns:nf="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="http://www.cisco.com/nxos:9.2.1.:configure_"
xmlns:m="http://www.cisco.com/nxos:9.2.1.:_exec"
xmlns:m1="http://www.cisco.com/nxos:9.2.1.:configure__if-eth-l2-non-member"
xmlns:m2="http://www.cisco.com/nxos:9.2.1.:configure__if-ethernet-switch" message-
id="1">
  <nf:edit-config>
    <nf:target>
      <nf:running/>
    </nf:target>
    <nf:config>
      <m:configure>
        <m:terminal>
          <interface>
            <__XML__PARAM__interface>
              <__XML__value>Ethernet1/10</__XML__value>
              <m1:switchport>
                <m1:mode>
                  <m1:__XML__PARAM__port_mode>
                    <m1:__XML__value>access</m1:__XML__value>
                  </m1:__XML__PARAM__port_mode>
                </m1:mode>
              </m1:switchport>
              <m2:switchport>
                <m2:access>
                  <m2:vlan>
                    <m2:__XML__PARAM__vlan-id-access>
                      <m2:__XML__value>100</m2:__XML__value>
                    </m2:__XML__PARAM__vlan-id-access>
                  </m2:vlan>
                </m2:access>
              </m2:switchport>
            </__XML__PARAM__interface>
          </interface>
        </m:terminal>
```

```
      </m:configure>
    </nf:config>
  </nf:edit-config>
</nf:rpc>
]]>]]>


sbx-n9kv-ao(xmlin)#
```

To use this tool, you enter the **xmlin** command. Inside the interactive tool, the switch prompt changes to hostname(xmlin)#. To enter configuration commands, you issue the command **config t**, and then, after each successful command, the switch outputs the message % Success. When all configuration commands have been entered, you issue the **end** command, and the switch immediately spits out the XML-encoded **<edit-config>** NETCONF rpc message that is equivalent to the commands you entered.

To perform the same process for a **show** command, you enter the **show** command at the **xmlin** interactive prompt, and when you press Enter, the tool immediately displays the XML-encoded NETCONF **<get>** RPC message, as shown in Example 17-16.

Example 17-16 *Using the **xmlin** Tool with a **show** Command*

```
sbx-n9kv-ao(xmlin)# show interface Eth1/10
<?xml version="1.0"?>
<nf:rpc xmlns:nf="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="http://www.cisco.com/nxos:9.2.1.:if_manager" message-id="1">
  <nf:get>
    <nf:filter type="subtree">
      <show>
        <interface>
          <__XML__PARAM__ifeth>
            <__XML__value>Ethernet1/10</__XML__value>
          </__XML__PARAM__ifeth>
        </interface>
      </show>
    </nf:filter>
  </nf:get>
</nf:rpc>
]]>]]>


% Success
sbx-n9kv-ao(xmlin)#
```

The **xmlin** tool converts commands to their corresponding operations as follows:

- A **show** command is converted to an rpc message that uses the **<get>** operation.
- Configuration commands are converted to an rpc message that uses the **<edit-config>** operation.
- An **exec** command is converted to an rpc message that uses the **<exec-command>** operation.

Finally, to exit the tool and return to the switch exec prompt, you use **exit**, as shown in Example 17-17.

Example 17-17 *Exiting the **xmlin** Tool*

```
sbx-n9kv-ao(xmlin)# exit
****************************************
****** Exited from the xmlin tool. *******
****************************************
sbx-n9kv-ao#
```

It is important to note here that the commands you enter in the **xmlin** interactive tool *are not actually applied to the switch*. This tool only shows the NETCONF equivalent of the commands you enter.

If you need the XML equivalent of a **show** command or of a **show** command's output, perhaps the easier and faster method would be to pipe the **show** command to **xmlin** for the XML-formatted NETCONF equivalent or to pipe the **show** command to **xmlout** for the XML-formatted command output. Example 17-18 shows examples of both operations.

Example 17-18 *Piping a **show** Command to **xmlin** and **xmlout** for the XML-Formatted Equivalents of the Command and Its Output*

```
sbx-n9kv-ao# show cdp global
Global CDP information:
    CDP enabled globally
    Refresh time is 60 seconds
```

```
    Hold time is 180 seconds
    CDPv2 advertisements is enabled
    DeviceID TLV in System-Name(Default) Format
sbx-n9kv-ao# sh cdp global | xmlin
<?xml version="1.0"?>
<nf:rpc xmlns:nf="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="http://www.cisco.com/nxos:9.2.1.:cdpd" message-id="1">
  <nf:get>
    <nf:filter type="subtree">
      <show>
        <cdp>
          <global/>
        </cdp>
      </show>
    </nf:filter>
  </nf:get>
</nf:rpc>
]]>]]>

% Success
sbx-n9kv-ao# sh cdp global | xmlout
<?xml version="1.0" encoding="ISO-8859-1"?>
<nf:rpc-reply xmlns="http://www.cisco.com/nxos:9.2.1.:cdpd" xmlns:nf="urn:ietf:params:x
ml:ns:netconf:base:1.0">
 <nf:data>
  <show>
   <cdp>
    <global>
     <__readonly__>
      <cdp_global_enabled>enabled</cdp_global_enabled>
      <refresh_time>60</refresh_time>
      <ttl>180</ttl>
      <v2_advertisement>enabled</v2_advertisement>
      <deviceid_format>DeviceID TLV in System-Name(Default) Format</deviceid_format>
     </__readonly__>
    </global>
   </cdp>
  </show>
 </nf:data>
</nf:rpc-reply>
]]>]]>
sbx-n9kv-ao#
```

As highlighted in Example 17-18, piping the output to **xmlin** results in an *rpc* message. Piping the same command output to **xmlout** results in an *rpc-reply* message.

Example 17-19 illustrates an XSD-based NETCONF session with a switch in which a hello message exchange is followed by a basic **show hostname** command.

Example 17-19 *Sample NETCONF Session via the XSD-Based API*

```
[kabuelenain@localhost ~]$ ssh -p 8181 admin@sbx-nxos-mgmt.cisco.com -s xmlagent


Welcome to the DevNet Always On Sandbox for Open NX-OS


This is a shared sandbox available for anyone to use to
test APIs, explore features, and test scripts.  Please
keep this in mind as you use it, and respect others use.
```

```
The following programmability features are already enabled:
  - NX-API
  - NETCONF, RESTCONF, gRPC
  - Native NX-OS and OpenConfig YANG Models

Thanks for stopping by.
Password:

! The switch hello message starts here
<?xml version="1.0" encoding="ISO-8859-1"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:url:1.0?scheme=file</capability>
    <capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.0</capability>
  </capabilities>
  <session-id>8544</session-id>
</hello>
]]>]]>

! The client hello message entered by the user starts here (can be copied and pasted to
the terminal)
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
  </capabilities>
</hello>
]]>]]>

! A get rpc message from the client to the device to get the device hostname (can be
copied and pasted to the terminal)
<nf:rpc xmlns:nf="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="http://www.cisco.com/nxos:9.2.1.:sysmgrcli" message-id="1">
  <nf:get>
    <nf:filter type="subtree">
      <show>
        <hostname/>
      </show>
    </nf:filter>
  </nf:get>
</nf:rpc>
]]>]]>

! The rpc-reply message from the device to the client showing the hostname
<?xml version="1.0" encoding="ISO-8859-1"?>
<nf:rpc-reply xmlns="http://www.cisco.com/nxos:9.2.1.:sysmgrcli"
xmlns:mod="http://www.cisco.com/nxos:9.2.1.:vdc_mgr" message-id="1"
```

```
xmlns:nf="urn:ietf:params:xml:ns:netconf:base:1.0">
 <nf:data>
  <mod:show>
   <mod:hostname>
    <mod:__readonly__>
     <mod:hostname>sbx-n9kv-ao</mod:hostname>
    </mod:__readonly__>
   </mod:hostname>
  </mod:show>
 </nf:data>
</nf:rpc-reply>
]]>]]>
```

When the SSH connection is established, the first thing received is the hello message, which lists the switch capabilities. A hello message listing the client capabilities is sent back to the switch. You can prepare this message beforehand and just copy and paste it into the SSH session. The hello message is followed by the NETCONF rpc message for the command **show hostname**. One possible way to generate this RPC message is by using the **xmlin** tool in another SSH session. At this point, the switch outputs the rpc-reply message that contains the switch hostname. Notice that all rpc and rpc-reply messages end in the six-character sequence]]>]]>, which signals the end of that particular message.

NETCONF can also be used with Python for more scalable automation. The de facto Python NETCONF client is ncclient. The ncclient library, which is not platform specific, is covered in Chapter 14 in detail.

NETCONF on IOS XR

As of this writing, the latest IOS XR version is 7.1.2, and the commands in this section apply to this version. However, these commands may change in later versions of the software.

To enable the NETCONF agent on IOS XR, the k9sec package must be installed, and crypto keys must be generated. Then, at a minimum, these two commands need to be executed: **netconf-yang agent ssh** and **ssh server netconf**. Then, optionally, you can use any of the following commands to configure the different parameters of the agent:

• **netconf-yang agent session limit** {*number-of-sessions*}: Limits the maximum number of allowed concurrent NETCONF sessions to the router. Allowed values are 1 to 1024, and no default limit is set.

• **netconf-yang agent session absolute-timeout** {*minutes*}: Sets the maximum allowed duration for a session, regardless of whether the session is idle. Allowed values are 1 to 1440 minutes, and no default value is set.

• **netconf-yang agent session idle-timeout** {*minutes*}: Sets the maximum allowed duration for a session to remain idle. Allowed values are 1 to 1440 minutes, and no default value is set.

• **netconf-yang agent rate-limit** {*Bps*}: Sets the maximum number of incoming bytes per second per session. Allowed values are 4096 to 4294967295, and there is no rate limiting by default.

• **ssh server netconf port** {*port-number*}: Sets the port number over which the NETCONF agent will be reachable. The default port is 830.

• **ssh server netconf vrf** {*vrf-name*}: Sets the VRF instance over which the NETCONF agent is reachable.

• **ssh server netconf ipv4 access-list** {*ipv4-access-list-name*}: Sets an IPv4 access list to restrict access to the netconf agent.

• **ssh server netconf ipv6 access-list** {*ipv6-access-list-name*}: Sets an ipv6 access-list to restrict access to the NETCONF agent.

• **ssh server capability netconf-xml**: Allows the NETCONF agent to be reachable over the default SSH port (port 22).

After the NETCONF agent is configured, the following commands allow you to check the operational status of the different components of the agent:

• **show netconf-yang capabilities**: Lists the device's NETCONF capabilities. This is the same list of capabilities that the router sends out in its hello message during the NETCONF session establishment.

• **show netconf-yang clients**: Displays a list of active client sessions, each identified by its session ID, the same session ID sent out by the device in its hello message to the client.

• **show netconf-yang notification subscriptions**: Displays information about the configured telemetry subscriptions.

• **show netconf-yang rate-limit**: Displays the rate-limiting statistics as well as the number of dropped bytes as a result of rate limiting.

• **show netconf-yang statistics**: Displays statistics per client session.

• **show netconf-yang status**: Displays the status of the NETCONF agent. The agent is running normally when the status is **ready**.

• **show netconf-yang trace**: Displays the log messages generated by the NETCONF agent.

You can obtain the YANG models supported by an IOS XR–based platform by using four different methods:

• Use the **show netconf-yang capabilities** command to list the device capabilities. The command lists the YANG models supported by the device.

• Send a **<get>** rpc message that utilizes the **<netconf-state>** element, as discussed in the section "The Content Layer" in Chapter 14. The device responds with an rpc-reply message listing the supported models.

• List all supported YANG models under the corresponding directory on the switch through the Bash shell.

• Go to the GitHub repo at https://github.com/YangModels/yang/tree/master/vendor/cisco/xr and download the modules for the specific IOS XR version running on your device. However, not all modules for a specific version are supported on all device models. For example, a model listed

under IOS XR Version 7.1.3 may be supported on NCS 5508 devices but not ASR9909 devices, even if both types of devices run IOS XR Version 7.1.3.

Cisco YANG modules follow the naming convention Cisco-IOS-XR-{*platform*}-{*technology*}-{*suffix*}.yang. Because several different platforms run IOS XR, the optional segment *platform* indicates the platform that this module applies to, with values such as **asr9k** or **ncs6k**. No *platform* segment indicates that the module applies to any platform running XR.

The *technology* segment indicates the technology that the module covers, such as **aaa-tacacs** or **ipv4-vrrp**.

The *suffix* segment takes one of four values:

• **cfg**: Modules defining configuration data models end with this suffix.

• **oper**: Modules defining operational data models end with this suffix. Operational submodules have the suffix oper-sub followed by an integer sequence number.

• **act**: Models defining YANG RPCs end in this suffix.

• **types**: Models defining data types not in the original YANG specification (for example, the BGP address-family datatype) end in this suffix.

For example, the module Cisco-IOS-XR-aaa-tacacs-cfg.yang defines a platform-agnostic model for the AAA/TACACS technology configuration data, and the Cisco-IOS-XR-asr9k-qos-oper-sub1.yang module is the first submodule of two submodules included (using an **include** statement) in the parent module Cisco-IOS-XR-asr9k-qos-oper.yang, which defines the data model for the QoS operational data for the ASR9K platform.

Platforms running Cisco IOS XR support candidate and running configuration datastores but not a startup configuration datastore. In other words, the **:startup** capability is not supported.

When you know how to enable NETCONF on IOS XR and fine-tune its parameters, and when you understand the naming conventions used for the Cisco YANG modules, working with NETCONF on IOS XR is no different than on any other platform.

NETCONF on IOS XE

As of this writing, the latest IOS XE version is (Amsterdam) 17.3.x. The commands in this section apply to this version. However, these commands may change in later versions of the software.

Devices running IOS XE generally expose a NETCONF API. Enabling NETCONF on IOS XE–based devices is as simple as issuing the CLI configuration command **netconf-yang**. To display the status of the NETCONF agent on the router, you use the command **show netconf-yang status**, as shown in Example 17-20.

Example 17-20 *Enabling NETCONF on IOS XE*

```
sr1000v-1(config)#netconf-yang


csr1000v-1#show netconf-yang status

netconf-yang: enabled

netconf-yang ssh port: 830

netconf-yang candidate-datastore: disabled
```

To optionally configure the SSH port for the NETCONF agent to a value other than the default 830, you can use the command **netconf-yang ssh port** {*port-number*}. You can also configure an IPv4 or IPv6 access list to limit the source IP addresses from which NETCONF messages may be received by using the command **netconf-yang** {**ipv4**|**ipv6**} **access-list name** {*access-list-name*}.

When NETCONF is enabled on IOS XE, the status of the different components of the protocol can be checked by using one of the following commands:

• **show netconf-yang datastores**: Lists the datastores that are enabled on the device and accessible via NETCONF.

• **show netconf-yang sessions** [**detail**]: Displays a list of active client sessions, each identified by its session ID, which is the same session ID sent out by the device in its hello message to the client. Adding **detail** to the end of the command displays also the statics per session.

• **show netconf-yang statistics**: Displays NETCONF statistics for all client sessions.

• **show netconf-yang status**: Displays the status of the NETCONF agent. The command output also shows whether the candidate configuration datastore is enabled.

• **show platform software yang-management process**: Lists the processes (daemons) required to run NETCONF and RESTCONF on the router and the status of each.

Example 17-21 shows the output of each of the five **show** commands with two active NETCONF sessions.

Example 17-21 *Checking the NETCONF Datastores, Sessions, Statistics, and Protocol Status*

```
csr1000v-1#show netconf-yang datastores

Datastore Name             : running

Globally Locked By Session : 25

Globally Locked Time       : 2020-02-01T09:55:10+00:00

Datastore Name             : candidate


csr1000v-1#show netconf-yang sessions

R: Global-lock on running datastore

C: Global-lock on candidate datastore

S: Global-lock on startup datastore
```

```
Number of sessions : 2

session-id  transport    username             source-host           global-lock
--------------------------------------------------------------------------------
20          netconf-ssh  developer            78.95.165.98          None
25          netconf-ssh  developer            78.95.165.98          R


csr1000v-1#show netconf-yang statistics
netconf-start-time  : 2020-02-01T09:26:22+00:00
in-rpcs             : 7
in-bad-rpcs         : 0
out-rpc-errors      : 1
out-notifications   : 0
in-sessions         : 7
dropped-sessions    : 0
in-bad-hellos       : 0


csr1000v-1#show netconf-yang status
netconf-yang: enabled
netconf-yang ssh port: 830
netconf-yang candidate-datastore: enabled


csr1000v-1#show platform software yang-management process
confd            : Running
nesd             : Running
syncfd           : Running
ncsshd           : Running
dmiauthd         : Running
nginx            : Running
ndbmand          : Running
pubd             : Running
```

Notice the letter **R** highlighted in the output of the command **show netconf-yang sessions**. This indicates that the session has a configuration lock on the running configuration datastore.

Most devices running IOS XE have all three datastores: running, startup, and candidate. However, as of this writing, NETCONF on IOS XE does not support the **:startup** capability, and therefore NETCONF messages cannot use **<startup/>**, neither as **<source>** nor **<target>**, in NETCONF messages.

The **:candidate** capability is supported on most platforms running IOS XE, but it is disabled by default, as you can see in Example 17-22.

Example 17-22 *NETCONF Support for the Candidate Datastore Is Disabled by Default on IOS XE Devices*

```
csr1000v-1#show netconf-yang status
netconf-yang: enabled
netconf-yang ssh port: 830
netconf-yang candidate-datastore: disabled
```

The candidate configuration datastore is enabled through the CLI by using the command **netconf-yang feature candidate-datastore**, as shown in Example 17-23.

Example 17-23 *Enabling NETCONF Support for the Candidate Datastore on IOS XE*

```
csr1000v-1(config)#netconf-yang feature candidate-datastore
netconf-yang and/or restconf is transitioning from running to candidate
netconf-yang and/or restconf will now be restarted, and any sessions in progress will be terminated


csr1000v-1#show netconf-yang status
netconf-yang: enabled
netconf-yang ssh port: 830
netconf-yang candidate-datastore: enabled
```

```
csr1000v-1#show netconf-yang datastores

Datastore Name             : running

Datastore Name             : candidate
```

When the candidate datastore is enabled, the **:writable-running** capability is not supported. Either of the capabilities may be supported at any point in time but not both. This means that after enabling the candidate configuration datastore, pushing configuration changes directly to the running configuration datastore through NETCONF is not allowed. This is evident from the list of capabilities sent back by the router on sessions started before and after enabling the candidate configuration datastore.

To terminate one of the NETCONF sessions from the CLI, you use the command **clear netconf-yang session** {*session-id*}. This command terminates the session identified in the command and releases any datastore locks this session had in place. This is the equivalent of a **<kill-session>** operation. To keep all sessions intact and just release a datastore lock, issue the command **clear configuration lock**. Example 17-24 shows the configuration lock held by session 25 released from the CLI.

Example 17-24 *Configuration Lock Release from the Switch CLI*

```
csr1000v-1#configure terminal

Configuration mode is locked by process '256' user 'NETCONF' from terminal '32132'.

Please try later.


csr1000v-1#clear configuration lock

Process <256> is holding the config session lock !

Do you want to clear the lock?[confirm]

csr1000v-1#configure terminal

Enter configuration commands, one per line.  End with CNTL/Z.

csr1000v-1(config)#
```

Now that you have learned about NETCONF on IOS XE, the NETCONF material covered in Chapter 14 comes into play. Note that the examples in this section are generated using the Cisco DevNet IOS XE sandbox.

Meraki

This section discusses the Meraki offering and the associated APIs that it exposes. For the examples, this section uses the Cisco Meraki always-on sandbox located at https://devnetsandbox.cisco.com/RM/Topology. At the time of this writing, the username for the sandbox is **devnetmeraki@cisco.com**, and the password is **ilovemeraki**. Please visit the sandbox web page for updated information.

Cisco Meraki is a complete ecosystem of products that provide an integrated connectivity solution for enterprises. Cisco Meraki incorporates hardware, software, and cloud services that work together to provide one or more of the following functions: wired LAN, wireless LAN, security, SD-WAN, cellular WAN, mobile device management (MDM), security video capture and streaming, traffic collection, and cloud-based network management to manage all these functions.

Cisco Meraki is a cloud-based solution. This means that all devices, regardless of their geographic location, are fully managed using a web application that connects to the Meraki cloud over the Internet. You can manage a switch or WAN device in your branch office or a security camera in another branch office in a different country by using a web portal that you have access to anywhere in the world.

Meraki is classified as a network management platform because, at the end of the day, Meraki boils down to the cloud-management component of the ecosystem. Meraki uses out-of-band cloud management. This means that only management traffic passes through the Meraki cloud, and user traffic never does.

Meraki APIs

Cisco Meraki provides a rich set of RESTful APIs that enable the integration of the Meraki solution with any third-party software, whether the software is a simple Python program that you wrote to extract a list of clients connected to a specific AP or a full network management solution from a vendor that needs to integrate with Meraki. Meraki exposes five APIs:

• **Dashboard API**: This is the API for retrieving configuration and state data and configuring the devices managed under the Meraki cloud. This is the primary API used for managing Meraki networks and devices, and it is the most frequently used API.

• **Webhook Alerts API**: This API provides push notifications. A service subscribes to these push notifications in order to receive an alert when an event takes place on the Meraki network.

• **Captive Portal API**: This API is used for configuring external services that facilitate guest access on wireless networks managed under the Meraki cloud, such as portals that register guest details before the guest is allowed access to the network.

• **Location Scanning API**: This API provides a programmable interface to track and locate clients on Meraki wireless LANs in a particular physical space.

• **MV Sense API**: This is the primary API for managing MV Series smart security cameras under the Meraki cloud.

Meraki, as a multitenant management solution for all Meraki customers, defines organizations as the highest level in the device hierarchy. Each organization is identified by a unique ID. Under each organization are one or more networks. Under each of these networks are the devices that belong to that network. The workflow for device provisioning in the Meraki cloud starts by acquiring device serial numbers and adding those numbers under a network that belongs to the organization.

Meraki Use Case: Dashboard API

The Dashboard API can be used to push configuration to devices in a Meraki network, whether to perform configuration management for existing devices or provision new devices and networks. The Dashboard API can also be used to retrieve configuration as well as state data.

The Dashboard API is a RESTful API that uses JSON for encoding. As of this writing, XML is not supported.

Before using the Dashboard API, you need to enable it. To do so, go to Organization on the left-hand side of the Meraki dashboard and then choose Settings. Scroll to the bottom of the screen, and you see a checkbox for enabling the API. Make sure this box is checked, as shown in Figure 17-6.

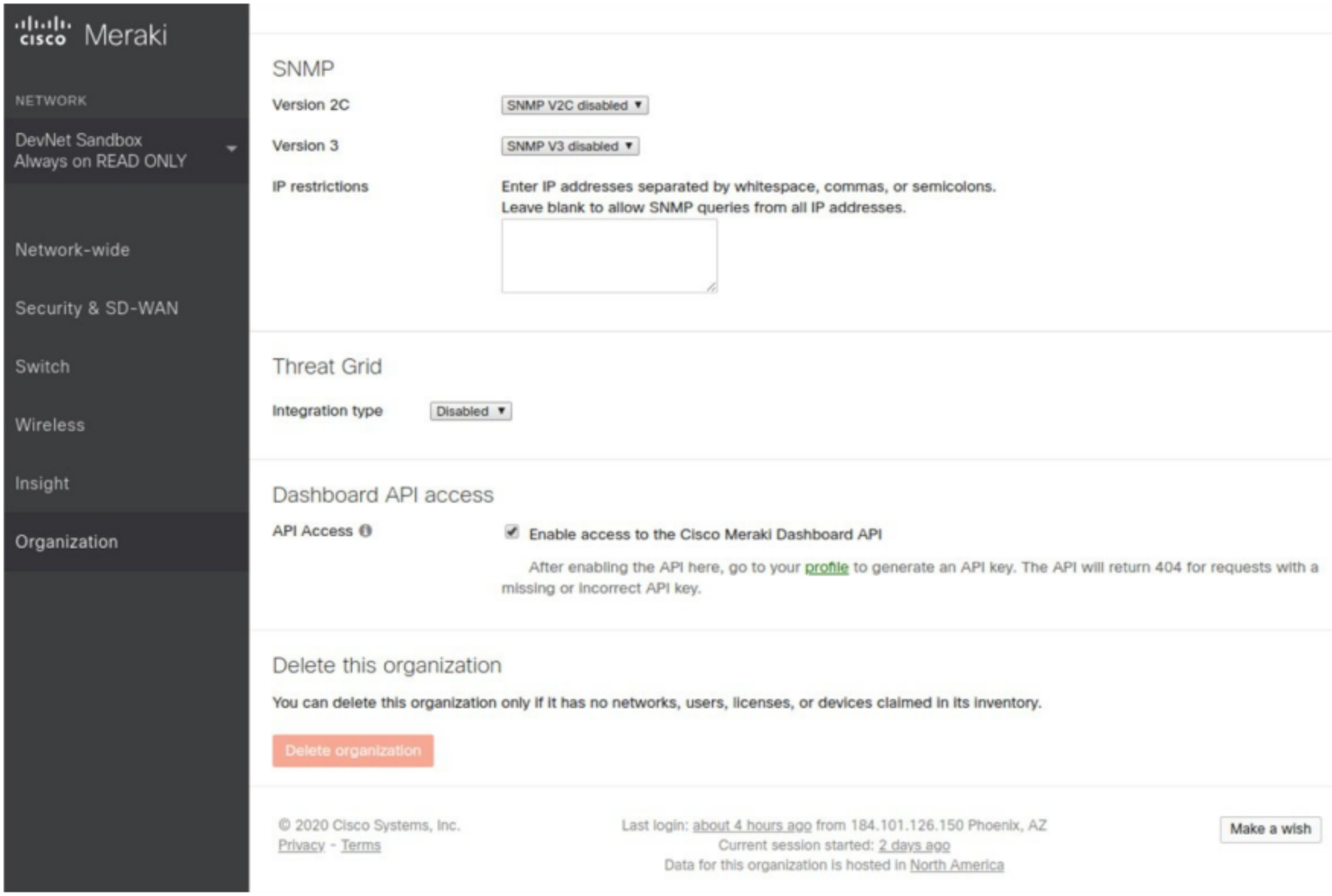

Figure 17-6 *Enabling the Dashboard API*

The next step is to generate an API key. This key identifies a particular administrator and will be used for the authentication and authorization of every HTTP request made to the Dashboard API by that administrator. This is done by adding a header named **X-Cisco-Meraki-API-Key** to each HTTP request and giving it a value equal to the API key. A maximum of two API keys per administrator are allowed.

The base URI for the Dashboard API is always https://api.meraki.com/api/v0. A resource's URI always starts with the base URI. The API version number identified by the /v0 segment at the end of the URI changes as new versions are released. (Refer to the API documentation at https://developer.cisco.com/meraki/api for the latest API specs.)

Using Postman or any other tool you feel comfortable with, retrieve the list of organizations on the Meraki sandbox by sending a **GET** request to the URI https://api.meraki.com/api/v0/organizations, as shown in Example 17-25.

Example 17-25 *Listing the Organizations Under the Meraki Cloud Using the Dashboard API*

```
! HTTP Request

GET /api/v0/organizations HTTP/1.1

X-Cisco-Meraki-API-Key: 6bec40cf957de430a6f1f2baa056b99a4fac9ea0

User-Agent: PostmanRuntime/7.22.0

Accept: */*

Cache-Control: no-cache

Accept-Encoding: gzip, deflate, br

Referer: https://api.meraki.com/api/v0/organizations

Connection: keep-alive


! HTTP Response

HTTP/1.1 200 OK

Server: nginx

Date: Tue, 11 Feb 2020 22:12:51 GMT

Content-Type: application/json; charset=utf-8

Transfer-Encoding: chunked

Connection: keep-alive
```

```
Vary: Accept-Encoding

Cache-Control: no-cache, no-store, max-age=0, must-revalidate

Pragma: no-cache

Expires: Fri, 01 Jan 1990 00:00:00 GMT

X-Frame-Options: sameorigin

X-Robots-Tag: none

X-UA-Compatible: IE-Edge,chrome=1

X-Request-Id: 912df4d8b45abf4ba16a71ce61320e31

X-Runtime: 0.235174

X-XSS-Protection: 1; mode=block

Content-Encoding: gzip


[
    {
        "id": "681155",
        "name": "DeLab",
        "url": "https://n6.meraki.com/o/49Gm_c/manage/organization/overview"
    },
    {
        "id": "566327653141842061",
        "name": "ENLabs",
        "url": "https://n6.meraki.com/o/iY6FHcg/manage/organization/overview"
    },
    {
        "id": "566327653141842188",
        "name": "DevNetAssoc",
        "url": "https://n6.meraki.com/o/dcGsWag/manage/organization/overview"
    },
    {
        "id": "646829496481089588",
        "name": "DevNetMultiDomainDemo",
        "url": "https://n149.meraki.com/o/rw48vavc/manage/organization/overview"
    },
    {
        "id": "549236",
        "name": "DevNet Sandbox",
        "url": "https://n149.meraki.com/o/-t35Mb/manage/organization/overview"
    },
    {
        "id": "52636",
        "name": "Forest City - Other",
        "url": "https://n42.meraki.com/o/E_utnd/manage/organization/overview"
    },
    {
        "id": "865776",
        "name": "Cisco Live US 2019",
        "url": "https://n22.meraki.com/o/CVQqTb/manage/organization/overview"
    },
    {
        "id": "463308",
        "name": "DevNet San Jose",
        "url": "https://n18.meraki.com/o/vB2D8a/manage/organization/overview"
    }
]
```

Notice the header X-Cisco-Meraki-API-Key and its value in the request, highlighted in Example 17-25.

To retrieve the information for one particular organization, you can use the URI https://api.meraki.com/api/v0/organizations/{*organizationid*}/, where you replace *organizationid* with the actual ID of that organization.

To list the networks under an organization, send a **GET** request to the URI https://api.meraki.com/api/v0/organizations/{*organizationid*}/networks. Example 17-26 shows the HTTP response, with the response body containing a list of networks under the organization whose ID is 549236 and name is DevNet Sandbox.

Example 17-26 *Listing the Networks Under Organization 549236 Using the Dashboard API*

```
HTTP/1.1 200 OK
Server: nginx
Date: Tue, 11 Feb 2020 22:20:29 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
Cache-Control: no-cache
Pragma: no-cache
Expires: Fri, 01 Jan 1990 00:00:00 GMT
X-Frame-Options: sameorigin
X-Robots-Tag: none
Last-Modified: Tue, 11 Feb 2020 22:20:29 GMT
X-UA-Compatible: IE=Edge,chrome=1
X-Request-Id: ab7da37bd159297208eb033bceb57c26
X-Runtime: 0.209099
X-XSS-Protection: 1; mode=block
Content-Encoding: gzip

[
    {
        "id": "L_646829496481104079",
        "organizationId": "549236",
        "name": "DevNet Sandbox Always on READ ONLY",
        "timeZone": "America/Los_Angeles",
        "tags": null,
        "productTypes": [
            "appliance",
            "switch",
            "wireless"
        ],
        "type": "combined",
        "disableMyMerakiCom": false,
        "disableRemoteStatusPage": true
    },
    {
        "id": "L_646829496481104279",
        "organizationId": "549236",
        "name": "DNENT3",
        "timeZone": "America/Los_Angeles",
        "tags": null,
        "productTypes": [
            "appliance",
            "camera",
            "switch",
            "wireless"
        ],
```

```
        "type": "combined",
        "disableMyMerakiCom": false,
        "disableRemoteStatusPage": true
    },

------ output truncated for brevity ------
```

To get the list of devices under a particular network, use the URI https://api.meraki.com/api/v0/networks/{*networkid*}/devices, as shown in Example 17-27 for network L_646829496481104079. (The HTTP request is omitted in this example.)

Example 17-27 *Listing the Devices Under Network L_646829496481104079 Using the Dashboard API*

```
HTTP/1.1 200 OK
Server: nginx
Date: Tue, 11 Feb 2020 22:33:23 GMT
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: Fri, 01 Jan 1990 00:00:00 GMT
X-Frame-Options: sameorigin
X-Robots-Tag: none
X-UA-Compatible: IE=Edge,chrome=1
X-Request-Id: c9767d4bcbec6e27ce5df0b8f7bcebb5
X-Runtime: 0.156155
X-XSS-Protection: 1; mode=block
Content-Encoding: gzip

[
    {
        "lat": 37.4180951010362,
        "lng": -122.098531723022,
        "address": "",
        "serial": "Q2QN-9J8L-SLPD",
        "mac": "e0:55:3d:17:d4:23",
        "wan1Ip": "10.10.10.106",
        "wan2Ip": null,
        "lanIp": "10.10.10.106",
        "networkId": "L_646829496481104079",
        "model": "MX65",
        "firmware": "wired-14-40",
        "floorPlanId": null
    },
    {
        "lat": 37.4180951010362,
        "lng": -122.098531723022,
        "address": "",
        "serial": "Q2HP-F5K5-R88R",
        "mac": "88:15:44:df:f3:af",
        "lanIp": "192.168.128.2",
        "networkId": "L_646829496481104079",
        "model": "MS220-8P",
        "switchProfileId": null,
        "firmware": "switch-11-22",
```

```
        "floorPlanId": null
    },
    {
        "lat": 37.4180951010362,
        "lng": -122.098531723022,
        "address": "",
        "serial": "Q2MD-BHHS-5FDL",
        "mac": "88:15:44:60:21:10",
        "lanIp": null,
        "networkId": "L_646829496481104079",
        "model": "MR53",
        "firmware": "wireless-25-14",
        "floorPlanId": null
    }
]
```

As you can see from Example17-27, this network is composed of three devices: an MX65 security and SD-WAN appliance, an MS220-8P switch, and an MR53 wireless access point.

Finally, to get a list of clients on a network, send a **GET** request to the URI https://api.meraki.com/api/v0/networks/{*networkid*}/clients.

Configuring a Meraki device is just as easy as retrieving information. You use the **POST** method to create resources, **PUT** to update resources, and **DELETE** to delete resources.

Before a VLAN can be created on a switch, network VLANs need to be enabled. To check whether VLANs are enabled for your network, send a **GET** request to the URI https://api.meraki.com/api/v0/networks/{*networkID*}/vlansEnabledState.

Example 17-28 shows the VLAN's enabled status on the network from Example 17-27. (The HTTP request message and the response headers have been omitted here since there is nothing new to show.)

Example 17-28 *Querying the VLAN's Enabled Status on the Network*

```
{
    "networkId": "L_646829496481104079",
    "enabled": false
}
```

You set the VLAN's enabled status to true by sending a **PUT** request to the same URI used in Example 17-28 ending in the segment /vlansEnabledState. The message body should be a JSON structure with the enabled field only set to **true**. (**networkId** is not required in the body in this case.)

Once the VLAN's enabled state is set to true, you may refer to the Meraki API documentation at https://developer.cisco.com/meraki/api/#!create-network-vlan for information on how to create a VLAN. For your convenience, Figure 17-7 shows this web page.

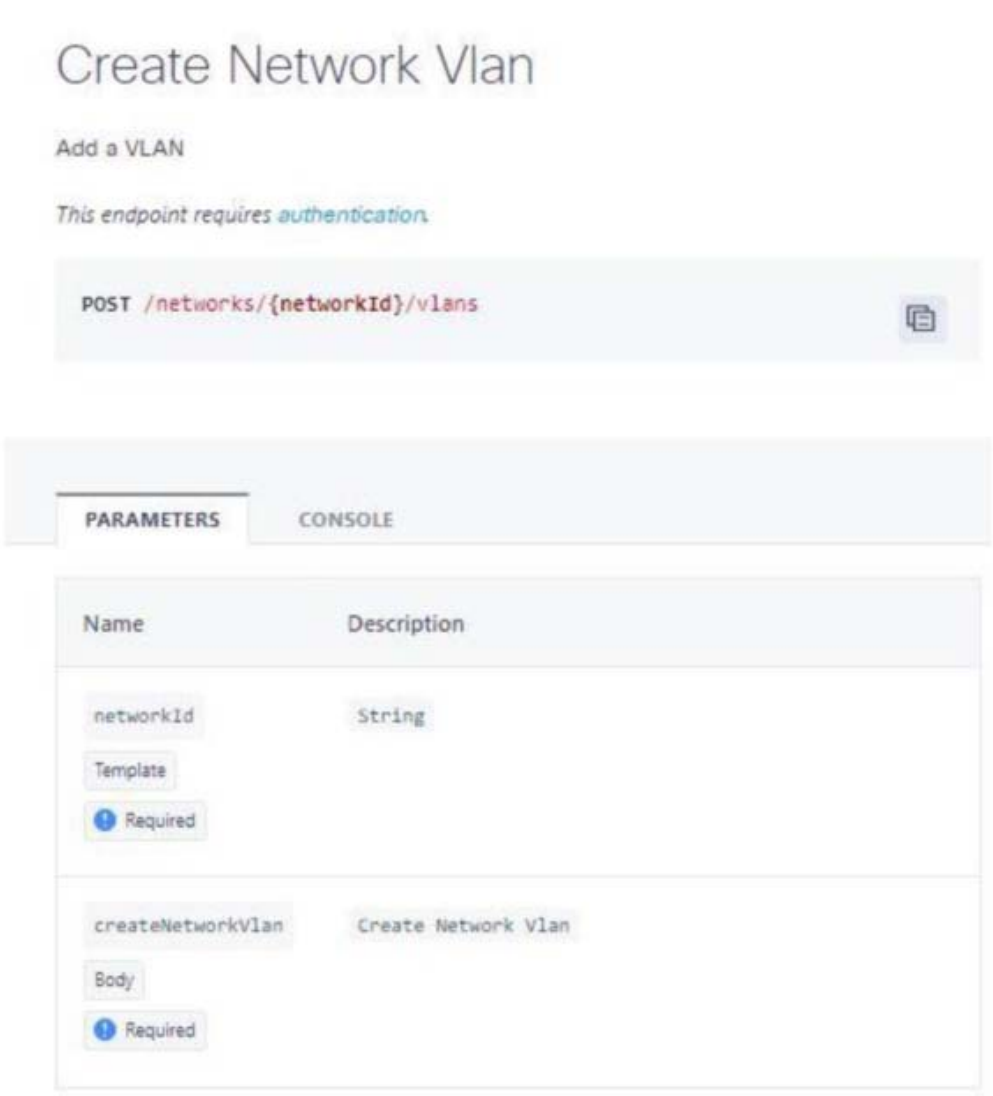

Figure 17-7 *API Documentation: Creating a Network VLAN*

As you can see, in order to create a VLAN, you need to send a **POST** request to the URI https://api.meraki.com/api/v0/networks/{*networkid*}/vlans, shown without the base URI in the documentation.

There are also two parameters, both of them shown as Required. The first parameter is the networkId, which is of type String and is a *Template* parameter, which means it goes into the URI. The second parameter is createNetworkVlan and is a *Body* parameter. If you hover your mouse over the phrase Create Network Vlan and click the mouse button, another webpage appears, listing the fields that should go into this parameter, as shown in Figure 17-8.

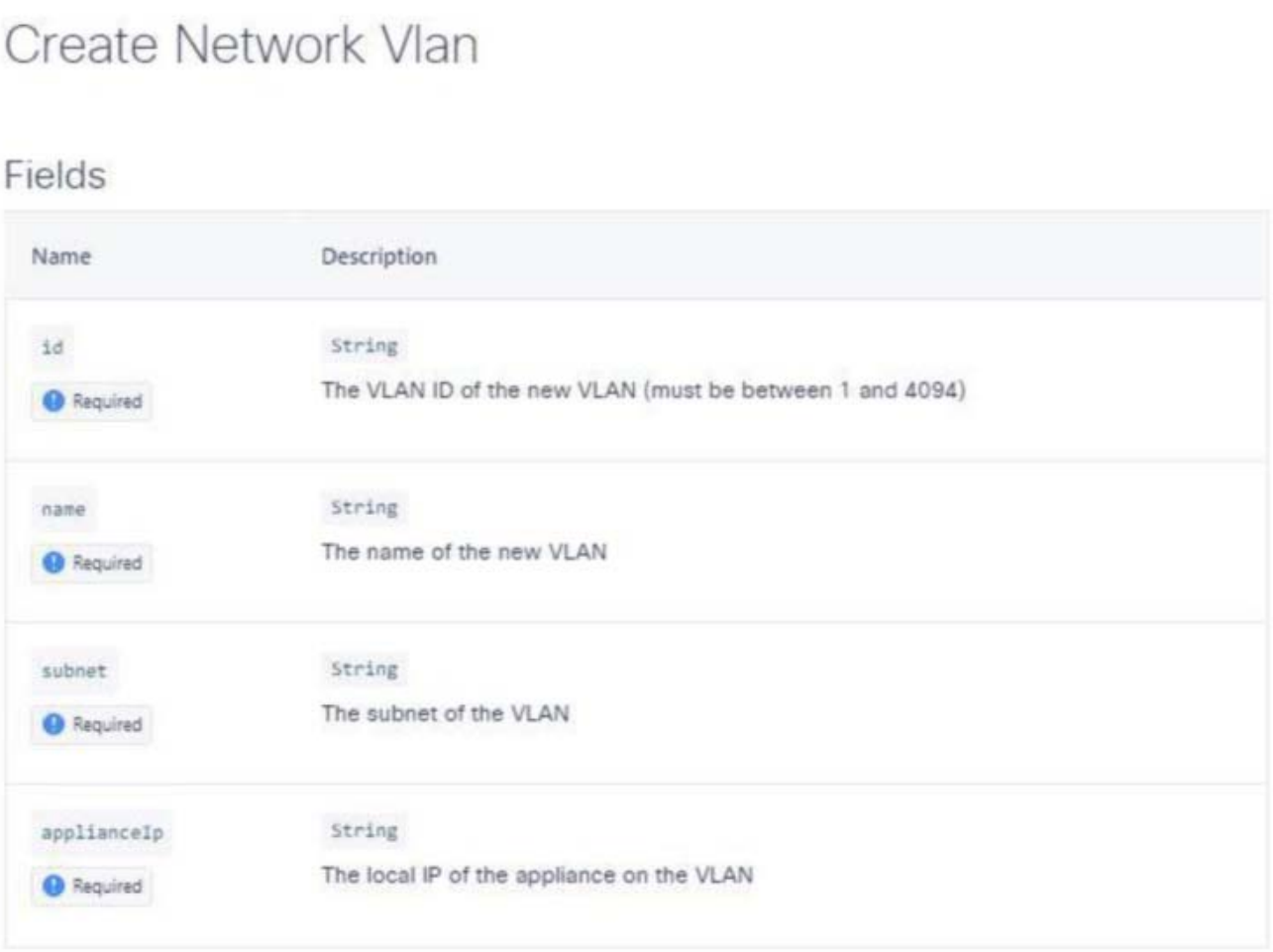

Figure 17-8 *API Documentation: Creating a Network VLAN and the Fields of the createNetworkVLAN Parameter*

The createNetworkVLAN parameter is a JSON structure that consists of four mandatory fields: id, name, subnet, and applianceIP. Therefore, to create VLAN 100, named DevTestVLAN, for subnet 10.0.1.0/24 on appliance 192.168.1.1 under network ID L_646829496481104079, you need to send a **POST** request to URI https://api.meraki.com/api/v0/networks/L_646829496481104079/vlans with the JSON structure in Example 17-29 as the message body.

Example 17-29 *HTTP Request Message Body to Create VLAN 100*

```
{
  "id": "100",
  "name": "DevTestVLAN",
  "subnet": "10.0.1.0/24",
  "applianceIp": "192.168.1.1"
}
```

DNA Center

This section describes Cisco DNA Center, its positioning in the automation ecosystem, and the APIs that it exposes.

Cisco Digital Network Architecture (DNA) is Cisco's architecture framework for intent-based networking (IBN). IBN is a new network management paradigm in which the outcomes required from a network are expressed in terms of business requirements (hence *intent*), and one or more products translate these requirements into actionable configuration, implement this configuration, and then monitor the outcome, amending the configuration as required, to keep the outcomes aligned with the business intent. IBN allows the network operator to focus on what needs to be accomplished rather than how to accomplish it.

DNA Center is a Cisco product that acts as the network management system, SDN controller, and analytics engine for the IBN ecosystem for an enterprise. DNA Center has the capability to manage Cisco and non-Cisco products, physical and virtual appliances, and fabric as well as standalone devices. Input to Cisco DNA Center is *intent* that DNA Center translates into configuration that gets pushed down to the relevant managed network devices. Network devices send back *context*, which is the state of the network devices, hosts, and traffic traversing the network and is used for both reporting analytics and assurance.

In DNA Center Version 2.1.2.x, when you log in to DNA Center, you see that the GUI has four sections: Overall Health Summary, Network Snapshot, Network Configuration, and Tools. Figure 17-9 shows the Network Configuration section.

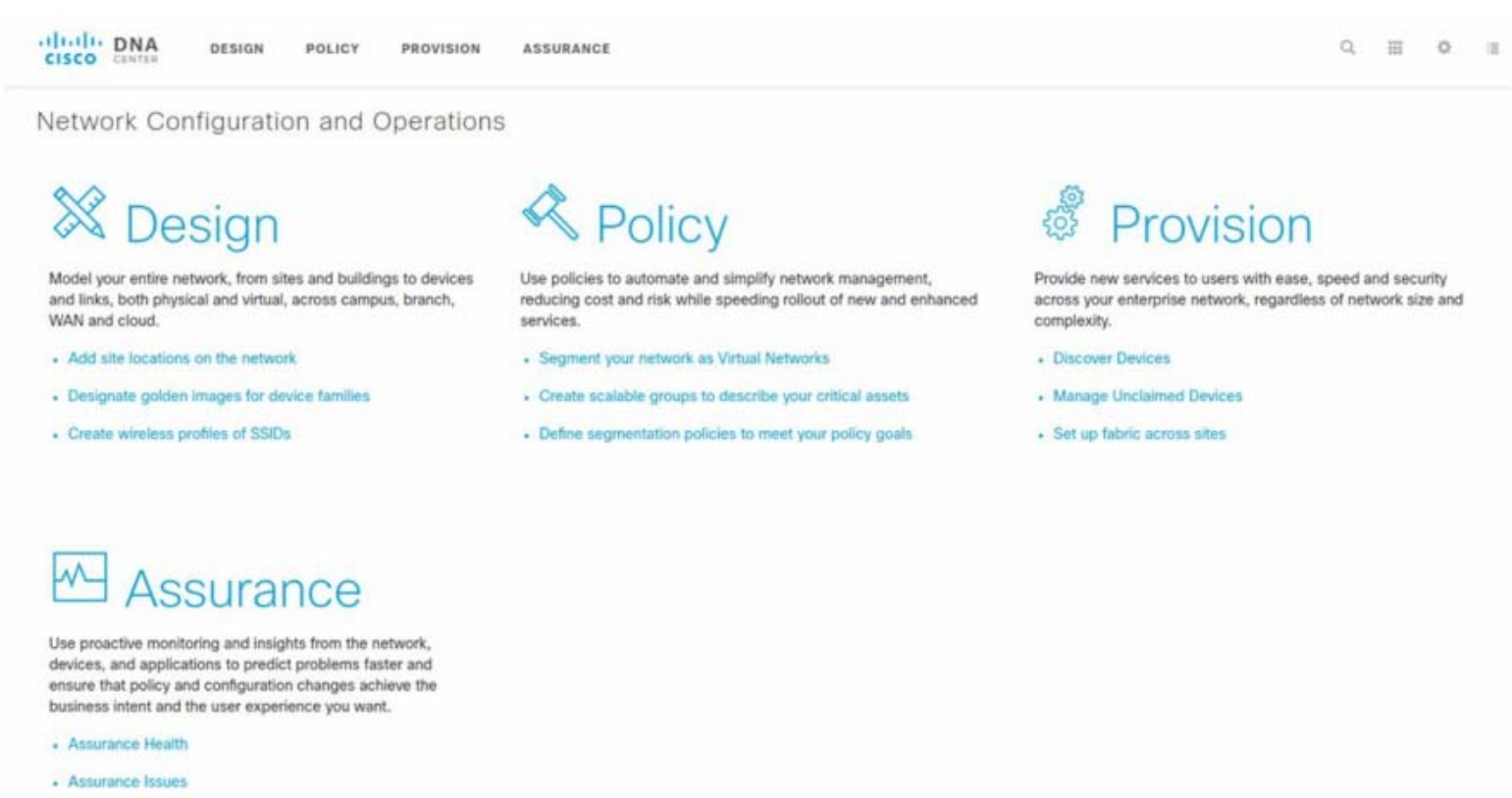

Figure 17-9 *A Section of the DNA Center GUI Showing the Different Aspects of Network Management*

In the Network Configuration section, you can see that DNA Center manages the network life cycle by managing four groups of network management tasks:

• **Design**: The tasks that you can complete from this section assist you with designing your network. You can define the geographic network hierarchy (country, state, city, building, floor, and so on), manage your network settings and services (DHCP, NTP and DNS servers, IP pools, wireless SSIDs and settings, and device credentials), manage the image repository for your devices, create network profile templates that will eventually be used to configure the wired and wireless network devices and fabrics that are managed by this DNA Center server, and manage the authentication templates for the authentication methods used for the different wired and wireless clients on the network.

• **Policy**: The tasks under this section are related to managing all the policies related to controlling application traffic, whether these policies are security policies or QoS policies. Under this section you also manage integration with the Cisco ISE server, if one exists in the network.

• **Provision**: This section allows you to complete tasks related to claiming network devices. Claiming a network device involves discovering the device and bringing it under the management supervision of this DNA Center server. This is also where you see and manage the inventory of claimed standalone and fabric devices.

• **Assurance**: In this section, you can monitor the performance of the network, applications, and hosts and manage the settings for corrective actions to take when network performance deviates from the business intent stated to DNA Center.

Note

Depending on the version of your DNA Center installation, the DNA Center GUI described thus far may be different from what you see in your environment. The general concepts remain the same, and the fundamental positioning and functions of DNA Center as a product remain the same. Remember that products and their APIs change and sometimes become end of life.

DNA Center APIs

DNA Center exposes a number of APIs that can be used to programmatically accomplish most of the tasks that would otherwise be done through the DNA Center GUI. The best way to describe the APIs or interfaces that DNA Center exposes is via their cardinal classification:

• **Northbound**: DNA Center exposes a RESTful API called the Intent API that allows an application to express business intent to DNA Center through an API call and have DNA Center interpret this business intent into a low-level configuration workflow to implement this intent. The Intent API can also be used to retrieve state and configuration data from DNA Center.

• **Southbound**: DNA Center can communicate with Cisco devices under its administration via several different interfaces, including the CLI, SNMP, and NETCONF. It also provides an SDK that can be used to develop device packages to be used to integrate with other vendors' equipment.

• **Eastbound**: DNA Center provides the facility for other systems to subscribe to event notifications generated by DNA Center. These notifications are sent to the subscribing systems via a push mechanism. This is done through a RESTful API commonly referred to as a *webhook*.

• **Westbound**: DNA Center exposes a RESTful API called the Integration API that allows ITSM, IPAM, reporting, and analytics systems to integrate with DNA Center.

The DNA Center API reference can be viewed at https://developer.cisco.com/docs/dna-center/.

Intent API

This northbound REST API uses the methods GET, POST, PUT, and DELETE and a JSON-encoded message body. This is the primary API used to communicate business intent to DNA Center, with the target of DNA Center converting this intent into actionable configuration and pushing it down to the relevant devices southbound. This API is also used to retrieve configuration and state information RESTfully from DNA

Center.

The Intent API is actually a massive and continuously growing interface in DNA Center. A finer classification scheme groups all the possible Intent API calls into functional domains and subdomains. For example, the subdomain Topology under the domain Know Your Network contains the API calls for retrieving the physical, layer 2, and layer 3 topology details for the network, in addition to the overall network health information. The subdomain Software Image Management (SWIM) under the domain Site Management contains the API calls for retrieving and managing the activation and distribution of software images. Figure 17-10 illustrates the Intent API domains and subdomains hierarchy.

Figure 17-10 *Intent API Domains and Subdomains*

As of DNA Center Version 2.1.2.x, there are a total of 7 domains and 27 subdomains. An updated list of all the possible Intent API calls for this version of DNA Center can be found in the Intent API reference at https://developer.cisco.com/docs/dna-center/#!cisco-dna-2-1-2-x-api-overview. In addition, the use case presented later in this section elaborates on the usage of Intent APIs.

Device Management

In the southbound direction, DNA Center communicates with the Cisco devices under its administration via the CLI (over SSH or Telnet), SNMP, or NETCONF. An SDK is also available to develop device packages to manage non-Cisco devices through DNA Center. The Cisco DNA Center SDK is based on the open-source Eclipse IDE.

Event Notifications and Webhooks

Much like most other programmable products from Cisco, DNA Center provides an interface through which third-party systems can subscribe to event notifications generated by DNA Center. The end receiving a notification from DNA Center is referred to as the *notification listener*. The notification listener may be an email server that receives an email notification from DNA Center. Alternatively, the listener may be an HTTP(S) server that receives the notification in the form of a **POST** HTTP(S) request message from DNA Center, with a JSON payload that contains the event details. These notifications in the form of HTTP(S) **POST** messages are called *webhooks*.

An event notification contains information about a specific event. Events are classified into domains and subdomains, which are the same as the ones used to classify Intent API calls (refer to Figure 17-10). This information, along with the *event attributes*, Type, Category, Severity, and Workflow, is included in the notification JSON body. Each of these fields can hold one of a number of predefined values, as listed in Table 17-1.

Table 17-1 *Event Attributes and Values*

Event Attribute	Allowed Values
Type	NETWORK, APP, SYSTEM, SECURITY, or INTEGRATIONS
Category	INFO, WARN, ALERT, ERROR, or TASK PROGRESS
Severity	1, 2, 3, 4, or 5
Workflow	Incident, Problem, Event, or RFC

A potential listener may subscribe to notifications for a particular event through the DNA Center GUI or by making an Intent API call by sending a **POST** request to the URI https://{*DNA-Center-Address-IP*}/dna/intent/api/v1/event/subscription with a JSON-encoded body that contains the details of the event that the listener wants to subscribe to. The event details in the JSON-encoded body may be obtained from the GUI or by making another API call by sending a **GET** request to the URI https://{*DNA-Center-Address-IP*}/dna/intent/api/v1/events?tags=ASSURANCE, which lists all the events that you may want to subscribe to.

Integration API

One of the principles on which DNA Center operates is *integration*. DNA Center was not developed with the intention of replacing all existing tools, but rather to integrate with other tools in order to optimize the existing processes and workflows. If an ITSM system, such as ServiceNow, is doing a great job handling, among other things, incident, change, and problem management, or if an IP address management (IPAM) system such as BlueCat is already managing the IP and DHCP pools in the network, then DNA Center should be able to integrate with those tools instead of trying to replace them.

DNA Center provides out-of-the-box seamless integration with a number of certified third-party tools, such as ServiceNow, BlueCat, Infoblox, and Tableau. Cisco created software modules called *bundles* that ship with DNA Center to enable the integration of DNA Center with a number of certified third-party tools; the bundles require very limited effort from a DNA Center administrator.

In the event that a non-certified third-party tool needs to integrate with DNA Center, an Integration REST API is provided that allows this integration.

Integrating DNA Center—whether with one of the certified third-party tools or some other tool—allows DNA Center to be part of the workflows defined on those tools. For example, DNA Center can be configured to request a ticket to be opened via ServiceNow if it detects a certain condition on the managed network. Alternatively, a change that is requested and approved through the change management workflow on ServiceNow could be fulfilled by DNA Center. An IP subnet that will be used by a group of hosts and is defined on DNA Center, could automatically be exported to Infoblox, along with the DHCP pool created for that subnet, without having to go to Infoblox and explicitly creating that DHCP pool.

Use Case: Intent API

This section shows three API calls to the Cisco DevNet DNA Center reservation-based sandbox. The first API call authenticates to the sandbox and gets an authorization token. The second API call retrieves the topology of the network, and the third retrieves a list of the devices managed by this instance of DNA Center. Each API call in this section is fully documented in the Intent API reference (see https://developer.cisco.com/docs/dna-center/#!cisco-dna-2-1-2-x-api-overview).

You can reserve the sandbox through the DevNet website, at https://devnetsandbox.cisco.com/RM/Topology. As of this writing, the sandbox is reachable at 10.10.20.85 using the username **admin** and password **Cisco1234!**. As you may have guessed from the private IP address of the sandbox, you need to connect through a VPN; this is the case for all reservation-based sandboxes on DevNet. The details of this connectivity are emailed to you when the time slot you reserve comes up. Both username and password on DNA Center are case sensitive. Make sure to visit the DevNet sandbox site for the latest sandbox details.

DNA Center authenticates each HTTP request sent to its Intent API by using an authorization token. You insert this token into each HTTP request by assigning it as the value of the header field named X-Auth-Token. In order to obtain this token, you need to make an API call to the URI https://10.10.20.85/dna/system/api/v1/auth/token, which is an Intent API call defined under the Authentication domain and subdomain.

Using Postman, you can create a new request and change the method to **POST**. Under the Authorization tab, choose Basic Auth from the Type drop-down list and then enter the username and password, each in its corresponding text box on the right (**admin**/**Cisco1234!** In this case). Click Send and then notice the response body at the bottom of the Postman window encoded in JSON, containing a Token object whose value is a string of characters.

Keep in mind that Basic Auth is nothing except the string {*username*}:{*password*}—in this case, admin:Cisco1234! encoded in Base64. This encoded Base64 string is then provided as the value of the Authorization header field in the request. This is done automatically for you in Postman. (Base64 encoding and HTTP authentication are covered in detail in Chapter 8, "Advanced HTTP.")

Now to use the token to retrieve useful information from DNA Center, you need to create a new request, this time using the default method, **GET**. Under the Headers tab, add a new header to your request by typing **X-Auth-Token** under Key and then copy and paste the Token value you got from the previous request into the Value field. Even better, you can create an environment variable and use that as the value for all your subsequent requests. When the token value expires, you request a new token and update it in one place only: the value of the environment variable. Postman is covered in detail in Chapter 7.

To retrieve the topology details of the network managed by this instance of DNA Center, add to the **GET** request you just created the URI https://10.10.20.85/dna/intent/api/v1/topology/site-topology. This URI is part of the Know Your Network domain and Topology subdomain. Example 17-30 shows the request message and part of the response message.

Example 17-30 *Listing the Sites Topology Defined in DNA Center Using an Intent API Call*

```
! HTTP Request

GET /dna/intent/api/v1/topology/site-topology HTTP/1.1

X-Auth-Token: eyJ0eXAiOiJKV1QiLCJhbGci...<Truncated>

User-Agent: PostmanRuntime/7.24.0

Accept: */*

Cache-Control: no-cache

Host: 10.10.20.85

Accept-Encoding: gzip, deflate, br

Connection: keep-alive


! HTTP Response

HTTP/1.1 200 OK

Content-Type: application/json;charset=utf-8

Transfer-Encoding: chunked

Connection: keep-alive

Date: Fri, 27 Mar 2020 17:20:05 GMT

Set-Cookie: JSESSIONID=hnppb8cevqz41nis3mfs29y6h;Path=/; Secure; HttpOnly

Expires: Thu, 01 Jan 1970 00:00:00 GMT

Server: webserver

Via: api-gateway

Cache-Control: no-store

Pragma: no-cache

Content-Security-Policy: default-src 'self' 'unsafe-inline' 'unsafe-eval' blob: data:

X-Content-Type-Options: nosniff

X-XSS-Protection: 1

Strict-Transport-Security: max-age=31536000; includeSubDomains

X-Frame-Options: SAMEORIGIN
```

```
{
    "response": {
        "sites": [
            {
                "id": "a7b75b75-eaee-4f7f-a2bb-53a4d36b6f6c",
                "name": "Building_1",
                "parentId": "1909e87e-75bc-4eb3-9494-b8e4e62f5773",
                "latitude": "-26.063795",
                "longitude": "28.082594",
                "locationType": "building",
                "locationAddress": "",
                "locationCountry": "South Africa",
                "displayName": "3213213",
                "groupNameHierarchy": "Global/South Africa/Gauteng/Woodlands/Building_1"
            },
            {
                "id": "1909e87e-75bc-4eb3-9494-b8e4e62f5773",
                "name": "Woodlands",
                "parentId": "b2ec99e3-5bbd-4bfa-8db4-1a7b6a805af3",
                "latitude": "",
                "longitude": "",
                "locationType": "area",
                "locationAddress": "",
                "locationCountry": "",
                "displayName": "3213212",
                "groupNameHierarchy": "Global/South Africa/Gauteng/Woodlands"
            },
            {
                "id": "acf4d799-f68e-41ac-a194-450c810defb3",
                "name": "Floor 1",
                "parentId": "a7b75b75-eaee-4f7f-a2bb-53a4d36b6f6c",
                "latitude": "",
                "longitude": "",
                "locationType": "floor",
                "locationAddress": "",
                "locationCountry": "",
                "displayName": "3213214",
                "groupNameHierarchy": "Global/South
Africa/Gauteng/Woodlands/Building_1/Floor 1"
            },
            {
                "id": "41c5c785-ba04-4fcd-bf40-f0452b3b662e",
                "name": "South Africa",
                "parentId": "33fbd22e-e408-4035-a5d4-53d91732b9f7",
                "latitude": "",
                "longitude": "",
                "locationType": "area",
                "locationAddress": "",
                "locationCountry": "",
                "displayName": "3213210",
                "groupNameHierarchy": "Global/South Africa"
            },
            {
```

```
                "id": "b2ec99e3-5bbd-4bfa-8db4-1a7b6a805af3",
                "name": "Gauteng",
                "parentId": "41c5c785-ba04-4fcd-bf40-f0452b3b662e",
                "latitude": "",
                "longitude": "",
                "locationType": "area",
                "locationAddress": "",
                "locationCountry": "",
                "displayName": "3213211",
                "groupNameHierarchy": "Global/South Africa/Gauteng"
            },
------ output omitted for brevity ------
        ]
    },
    "version": "1.0"
}
```

Notice the usage of the authorization token as the value of the X-Auth-Token header field, highlighted in Example 17-30. The HTTP response is a JSON object listing a number of sites. The site hierarchy is not immediately noticeable. However, note the parameter named groupNameHierarchy, which shows you the hierarchy and the relationships between the sites.

To retrieve a list of network devices claimed by the DNA Center sandbox, send a **GET** request to the URI https://10.10.20.85/dna/intent/api/v1/network-device. Due to the length of the response received, Example 17-31 shows the part of the response for one device only.

Example 17-31 *Listing the Devices Managed by DNA Center Using an Intent API Call*

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Server: webserver
Set-Cookie: JSESSIONID=4CC491036A0560A78A1E8D77D1FFC60D; Path=/apic-em-inventory-
manager-service; HttpOnly; Secure; HttpOnly
Date: Fri, 27 Mar 2020 17:21:41 GMT
Via: api-gateway
Cache-Control: no-store
Pragma: no-cache
Content-Security-Policy: default-src 'self' 'unsafe-inline' 'unsafe-eval' blob: data:
X-Content-Type-Options: nosniff
X-XSS-Protection: 1
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Frame-Options: SAMEORIGIN


{
    "response": [
        {
            "memorySize": "3735220224",
            "family": "Wireless Controller",
            "type": "Cisco 3504 Wireless LAN Controller",
            "lastUpdated": "2020-03-27 12:27:31",
            "lineCardCount": "0",
            "lineCardId": "",
            "locationName": null,
            "managementIpAddress": "10.10.20.51",
            "platformId": "AIR-CT3504-K9",
            "reachabilityFailureReason": "",
            "reachabilityStatus": "Reachable",
```

```
            "series": "Cisco 3500 Series Wireless LAN Controller",
            "snmpContact": "",
            "snmpLocation": "",
            "tagCount": "0",
            "tunnelUdpPort": "16666",
            "waasDeviceMode": null,
            "apManagerInterfaceIp": "",
            "associatedWlcIp": "",
            "bootDateTime": "2019-04-10 00:53:31",
            "collectionStatus": "Managed",
            "errorCode": null,
            "errorDescription": null,
            "interfaceCount": "0",
            "roleSource": "AUTO",
            "lastUpdateTime": 1585312051211,
            "upTime": "352 days, 11:34:12.00",
            "serialNumber": "FCW2219M007",
            "macAddress": "50:61:bf:57:51:00",
            "collectionInterval": "Global Default",
            "inventoryStatusDetail": "<status><general code=\"SUCCESS\"/></status>",
            "hostname": "Cisco_57:51:02",
            "deviceSupportLevel": "Supported",
            "softwareType": "Cisco Controller",
            "softwareVersion": "8.8.111.0",
            "location": null,
            "role": "ACCESS",
            "instanceTenantId": "5be5fcaaee9a67004cf94d6b",
            "instanceUuid": "aff8a3e1-4462-4e39-b974-40ea16b609e8",
            "id": "aff8a3e1-4462-4e39-b974-40ea16b609e8"
        },
------ output omitted for brevity ------
    ],
    "version": "1.0"
}
```


Retrieving the full list of devices every time you make an API call may not be very practical at times. Looking at the API documentation, under the Devices subdomain, you can see that the list of devices retrieved can be filtered by any one of several criterion, one of which is the hostname. To retrieve the details of the device whose hostname is Cisco_57:51:02, all you need to do is send a **GET** request to the URI https://10.10.20.85/dna/intent/api/v1/network-device?hostname=Cisco_57:51:02.

Collaboration Platforms

This section covers the programmability and automation features of Cisco's collaboration line of products, which fall into four categories: Unified Communications, Contact Center, Conferencing, and Collaboration Endpoints.

Cisco's Collaboration Portfolio

Cisco's collaboration portfolio includes the following:

• **Unified Communications**: These products provide converged solutions for voice, video, data, and mobile communication. Products in this category include the following:

• **Cisco Webex Teams**: This solution from Cisco has the feel of a messenger or chatting application but is actually a central point for collaboration of different teams. From Webex Teams you can send text messages, share files, whiteboard, start Webex meetings, and more. Webex Teams revolves around the concepts of People (registered users of Webex Teams), Teams (groups of people with a set of rooms that are visible to all members of that team), and Spaces or Rooms (virtual meeting places where people post messages and collaborate to get work done). Several other objects, such as Devices, Events, and Places, exist in Webex Teams, but they act as the backdrop for the former three primary objects.

• **Cisco Unified Communications Manager (CUCM)**: This is the base software product from Cisco that provides the infrastructure required for IP telephony, high-definition video, unified messaging, instant messaging and presence. It is leveraged by other Cisco products, such as the Business Edition.

• **Cisco Webex Cloud Calling**: This is a full cloud solution that provides the same functionality as on-premises CUCM, in addition to all the benefits and functionality provided by native cloud solutions.

• **Cisco Business Edition**: This is a line of products, comprised of integrated software and hardware, that provide collaboration solutions targeting small to enterprise-grade businesses. Each solution in the Business Edition line provides different capacity, scalability, high-

availability, and management options (for example, cloud versus on-premises management). This solution leverages several existing products and solutions from Cisco, including Cisco's UCS servers, CUCM, and Contact Center Express software.

• **Contact Center**: These products provide a contact center solution for customers who need to interact with their customers, whether internal or external, and provide a form of support. Contact Center expands on the function of legacy call centers in that agents interact with customers not only over the phone but via online chat, social media, SMS, and other messaging applications. Products under this category include the following:

• **Cisco Unified Contact Center (Express or Enterprise)**: This is the base software product from Cisco that provides the infrastructure for the Contact Center functionality. The Express offering (CCX) provides support for up to 400 agents, and the Enterprise offering (UCCE) supports up to 24,000 agents. This software provides different channels to support next-generation Contact Center channels to communicate with customers, such as inbound voice, outbound voice, outbound IVR, and digital channels. The Enterprise offering goes as far as supporting features such as post-call IVR, email, and web intercept surveys.

• **Cisco Webex Contact Center (Enterprise)**: This is the native cloud Contact Center offering from Cisco. A cloud-based solution provides integrated analytics and artificial intelligence in addition to rapid deployment and minimal capital and operational cost, since the whole solution is hosted in the Cisco cloud. Webex-based cloud solutions also provide seamless integration with other Webex products, such as Webex Teams.

• **Cisco Finesse**: Through a Web 2.0 interface, Finesse provides agent and supervisor desktops in a contact center. Finesse integrates with the other products in the collaboration portfolio to provide the full contact center functionality.

• **Conferencing**: These products provide next-generation video conferencing solutions. Products in this category include the following:

• **Cisco Webex Meetings**: This cloud-based software solution enables scheduling, running, and recording of meetings involving voice, video, messaging, whiteboarding, screen sharing, and other functions. It runs seamlessly on a number of devices, including PCs, mobiles phones, and Cisco endpoint devices.

• **Cisco Webex Support**: This cloud-based solution enables support teams to provide remote support to their customers by facilitating tasks such gaining remote access to a customer PC. This solution requires the installation of an agent on the remote machine that requires support.

• **Cisco Meeting Server**: This is an on-premises conferencing solution offering from Cisco.

Other services included in this category are Webex Edge, Webex Events, Webex Training, and Webex Webcasting.

• **Collaboration Endpoints**: These are the physical endpoint appliances provided by Cisco. Products in this category include the following:

• **Cisco Webex Board**: This is a digital whiteboard that connects wirelessly and is touch-based. It is used for meeting room presentations and integrates with the Webex line of products to extend its functionality to support conferencing. As of this writing, it comes in three sizes.

• **Cisco Webex Room Series**: This is a group of products that are geared toward running Webex Meetings in conference room settings. The product line includes the Room products, which are 4K screens fitted with microphones and cameras, and Room Kits, which are used to transform third-party 4K displays into full Webex Meetings endpoints. The Room series of products runs the programmable RoomOS software.

• **Cisco IP Phones**: This is a line of IP phones with a range of capabilities that come in both wired and wireless forms.

Collaboration APIs

Most products in Cisco's collaboration portfolio are programmable and expose a number of APIs that enable programmable management of the devices and solutions as well as integration with third-party systems. This section covers the programmability features and APIs exposed by CUCM, Webex Meetings, Webex Teams, Webex Devices, and Finesse.

Cisco Unified Communications Manager (CUCM)

CUCM exposes five APIs:

• **Administrative XML Web Service (AXL) API**: This API is used to perform CRUD operations on CUCM objects. This API is SOAP-based and uses HTTP 1.0 Basic Authentication. AXL message bodies are encoded in XML and use Web Services Description Language (WSDL) and XML Schema Definitions (XSD). AXL is the primary API used to manage CUCM programmatically and can be used to provision, retrieve the status of, update, or delete objects on the CUCM, such as phones, users, device pools, and dial plans.

• **Cisco Emergency Responder (CER) API**: The CER is an emergency communications system that manages 911 and similar emergency calls from one of the devices under CUCM administration. The CER routes emergency calls as required in addition to providing extra information, such as the caller's location. CER also automatically stores a call log of all emergency calls. The CER API, which provides programmatic access to Cisco CER, is a REST API that uses HTTP Basic Authentication and only supports the **GET** method and an empty message body in the requests. The message bodies of HTTP responses from this API are encoded in XML. The CER API allows third-party applications to integrate with and leverage the services provided by CER.

• **Platform Administrative Web Services (PAWS) API**: This API is meant for programmatically managing CUCM clusters. This new API is available only on Versions 9.0(1) and later. This API can be used to retrieve CUCM software version numbers or hardware details. It can also be used to upgrade the CUCM or reboot it. This is an XML/SOAP-based API.

• **CUCM Serviceability API**: CUCM Serviceability is a group of services and tools that are geared toward monitoring the status of CUCM as well as reporting, diagnosing, and resolving issues. The CUCM Serviceability API provides programmatic access to these tools via an XML/SOAP interface. Examples of data that can be retrieved from these tools includes the number of phones and devices registered to CUCM and the connection status of these endpoints. You may also retrieve the number of concurrent connections to the CUCM TFTP server (indicating the number of devices downloading new firmware) and CUCM's performance during these downloads. This API can also be used to retrieve call detail record (CDR) and CUCM logs.

• **User Data Services (UDS) API**: This API is intended for use by a specific authenticated user to manage their own experience and settings on CUCM. Calls to the UDS API may be made to retrieve general information common to all users, such as calls to do directory searches for other users, to retrieve the CUCM time zones or to retrieve the list of nodes in the CUCM cluster. Or the calls to the API may be done to manage this user's speed dial settings or credentials. The UDS API is a REST API that uses the methods **GET**, **PUT**, **POST**, and **DELETE**.

Webex Meetings

Webex Meetings exposes four APIs:

• **XML API**: This API uses the **POST** method and an XML-encoded body to provide an interface to integrate custom applications with Webex Meetings. This is the primary API to use to integrate with Webex Meetings. For example, if you would like to schedule a Webex meeting, get the link to a meeting, delete a meeting, or list summary information for scheduled meetings, this is the API to call. But Meeting services is only

one part of the functionality exposed by this API. This API can also be used to manage Webex Training, Webex Events, and Webex Support Services, among other functions.

• **URL API**: This API uses HTTPS requests with URIs containing PHP calls in the form of parameters. These PHP calls initiate service requests on the Webex Meetings server. The URL API provides a very limited number of functions and is often used as a lightweight alternative to the XML API.

• **Teleconference Service Provider (TSP) API**: This API provides external TSPs an interface to integrate their teleconferencing service with Cisco Webex Meetings.

• **REST API**: Cisco also provides a newly developed REST API, which will eventually expose all functionality of Webex Meetings currently exposed by the XML API.

Webex Teams

Webex Teams exposes a REST API that provides a programmatic interface to Webex Teams. Using this API, you can list, create, and delete rooms (spaces), teams, and messages and manage people and their memberships. Authentication to this API involves using an authentication token as the value of the Authorization header in the HTTP request. This token is generated by creating a developer account on https://developer.webex.com/.

This API uses the methods **GET**, **PUT**, **POST**, and **DELETE**, and the request message body is encoded in application/json or application/x-www-form-urlencoded; the responses are always encoded in application/json. The use case later in this section uses this REST API to create a new space, add participants to the space, and send messages to the space.

The Webex Teams REST API also supports webhooks. You can create a webhook so that a custom application you created receives a notification (which is actually an HTTP POST request) when a particular event occurs in Webex Teams.

Webex Devices

The majority of Webex endpoints run software called Cisco Collaboration Endpoint Software, which exposes an API called *xAPI* (short for *Experience API*). xAPI can be used to configure these endpoints or to integrate them with third-party control systems, such as Crestron, AMX/Harman, or Extron.

xAPI uses a variety of transports, each requiring its own configuration on the GUI of the software in order to be enabled. The API calls can made over SSH, Telnet, HTTP/HTTPS, WebSocket, or a serial connection. The encoding of the input and output to and from xAPI is configurable to Terminal (CLI style line-based), XML, or JSON, with Terminal being the default.

Commands to xAPI are classified into several categories. These are the main ones:

• **xCommand**: These commands direct the device to execute one or more actions, such as to join a Webex meeting, dial a phone number, or mute the microphone.

• **xConfiguration**: These commands target the device settings, such as the time zone or the IP address of the device, or the default audio level of the speaker on the device.

• **xStatus**: These commands retrieve the current status of the device, such as the current device speaker volume in dB.

• **xFeedback**: These commands specify what parts of the configuration and status hierarchies to monitor.

• **xPreferences**: These commands are used to set preferences for RS-232, Telnet, and SSH sessions.

Finesse

Cisco Finesse exposes four REST APIs. All of these APIs encode the message body, if one exists, in XML. Finesse APIs either use HTTP Basic Authentication using a Base64-encoded authorization header, or a bearer token.

Finesse Desktop APIs are used by the Finesse desktops (for both agents and supervisors) to communicate with the Cisco Finesse server or the Cisco Unified Contact Center (Enterprise or Express). The API is used for bidirectional communication of information related to the different Contact Center entities. The API calls are further broken down into the following categories:

• **User**: These are the agents and supervisors. Each user is represented by a User object that holds the information of that user, such as the first and last names, login ID and name, role, team, and status (such as logged in and ready to take calls). API calls in this category are used to sign in or sign out agents and communicate agent state information to the Finesse server.

• **Dialog**: API calls in this category are used to communicate information related to voice calls with customers and non-voice tasks, together known as *dialogs*. (Recall the difference between a legacy call center and a next-generation contact center?) Each dialog is represented by a Dialog object that holds the information of that particular dialog.

• **Queue**: API calls in this category are used to communicate information related to call queues and the statistics for those queues, such as the number of calls in queue and the start time of the longest call in queue. Each queue is represented by a Queue object that is used to hold the information for that queue.

• **Team**: A team is a group of users, such as a group of agents assigned to a specific technology or a group of agents at a certain level of the escalation hierarchy. Each team is represented by a Team object that contains information on that team, including the users assigned to the team. The API calls in this category are used to communicate team information.

• **Client Log**: The APIs in this category are used to post client-side logs to the Finesse server. These APIs are one-way only.

• **Single Sign-On**: The Finesse desktop or any other third-party desktop application that needs to integrate with the Finesse server use the APIs in this category to manage SSO token-related operations.

• **Team Message**: The APIs in this category can be used to manage (send and retrieve) messages sent to all the users in one or more teams. This API is used by a supervisor or the Finesse server administrator.

Configuration APIs are used by Finesse administrators to configure the different Finesse entities, such as system, cluster, and database settings; reason codes and wrap-up reasons; phonebooks and contacts; team resources; and workflows and workflow actions.

Serviceability APIs are used to retrieve system information such as the installed licenses and deployment type (with CCE or CCX), retrieve diagnostic and performance data, or retrieve runtime information, such as the number of logged-in agents.

And finally, the Cisco Finesse Notifications service provides subscription-based webhook-style notifications.

Use Case: Webex Teams

This section shows how to use the Webex Teams REST API to discover the user called *networkdeveloper*, whose email is networkdeveloper@thenetdev.com. Then you will create a team called *Webex Teams Devs TEAM (via REST API)* and then add networkdeveloper to the team. You will then create a space (room) named *Webex Teams Devs ROOM/SPACE (via REST API)*, attached to the team you just created. You will then add the user networkdeveloper to the room. Finally, you will send a message to that room. All this is accomplished solely via the Webex Teams REST API.

As stated in the previous section, you need to have an authentication token sent with every HTTP request for authorizing that request. You can get a temporary token that is valid for 12 hours by logging in to https://developer.webex.com. If you don't have a Webex Developer account, go ahead and create one. It takes only a few minutes.

When you are logged in, click the Documentation link at the top of the page and then click API Reference on the left side of the page. Scroll down and click the People section. A list of all API calls under this category appears on the right side of the page. Click the List People API: the first API in the list that uses the method **GET** and the URI https://api.ciscospark.com/v1/people. You are then redirected to the API documentation for that specific API call.

To the right side of each documentation page, you can see the title Header with the word Authorization under it, as shown in Figure 17-11. This is the value of the token that has to be included in each HTTP request you send. If you will use Postman, you need to add a header field in the Request Headers section with a key equal to **Authorization** and value equal to **Bearer** {*token_value*}. For the *token_value*, just click the Copy button to the right of the hidden value on the documentation page. Note that the token is valid for 12 hours or until you log off from the developer.webex.com site.

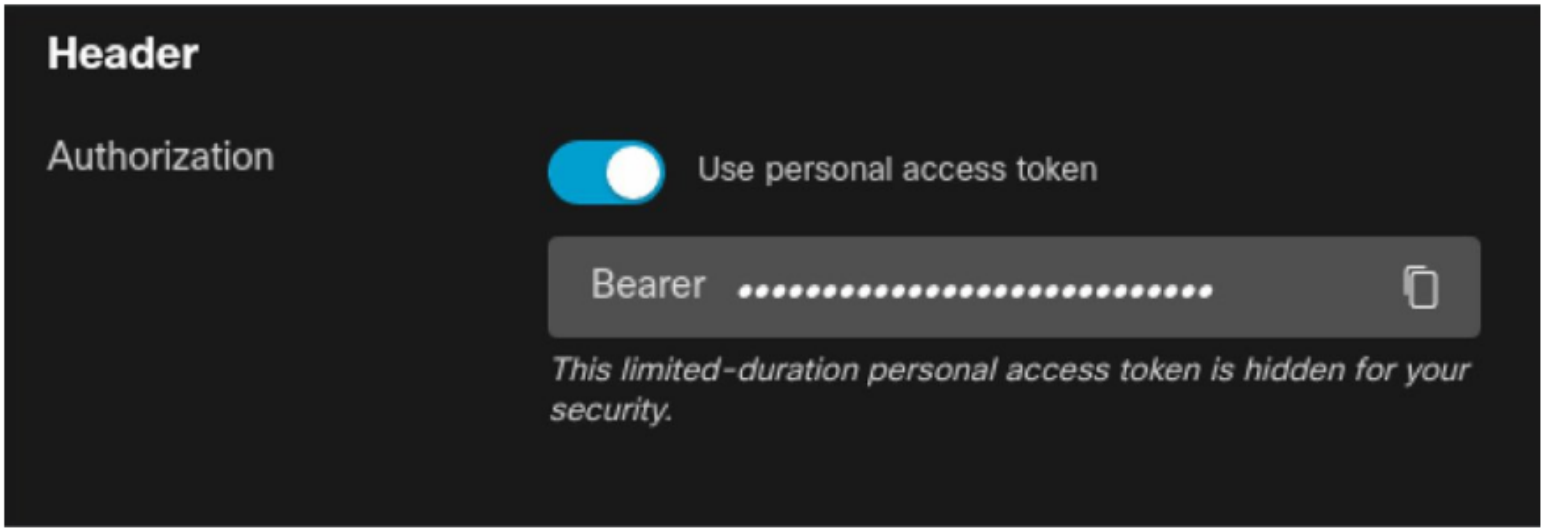

Figure 17-11 *Authorization Header Value on the API Reference Page*

API calls can be made through the API Reference web page itself or through any other HTTP client, such as Postman or cURL. If you opt for testing the API call through the page, simply fill in the fields on the right side of the page and click the orange Run button at the bottom of the page. Mandatory parameters are marked as Required under the parameter name. For this use case, Postman is a good option as it allows you to fill in all values manually and inspect the actual requests and responses in their raw format from the Postman Console.

Now open Postman and create a new request. Use the default method, **GET**, and enter the URI https://api.ciscospark.com/v1/people?email=networkdeveloper%40thenetdev.com. In the Headers section add the authorization header and give it the value of the token from the API reference page. The message body will be empty for this API call. Note that the email of the user that you are querying is added to the URI as the value of the email parameter, as described in the API reference documentation. The HTTP request and response are shown in Example 17-32.

Note

If you need a refresher on Postman or URI query parameters, review Chapter 7. For more on HTTP authorization, review Chapter 8.

Example 17-32 *Using a **GET** Request to Query the User networkdeveloper by Email*

```
! HTTP Request

GET /v1/people?email=networkdeveloper%40thenetdev.com HTTP/1.1

Authorization: Bearer

Y2MzN2IwM2YtMWY4MS00MTVhLTk2NzQtNzcwZDdmYWQ1ZTI3ZjQwMDExODEtMTM4_PF84_consumer

User-Agent: PostmanRuntime/7.24.0

Accept: */*

Cache-Control: no-cache

Host: api.ciscospark.com

Accept-Encoding: gzip, deflate, br

Connection: keep-alive


! HTTP Response

HTTP/1.1 200 OK

Cache-Control: no-cache, no-store

Via: 1.1 linkerd
```

```
Transfer-Encoding: chunked

Content-Encoding: gzip

TrackingID: ROUTER_5E825706-09B6-01BB-010F-5DA89168010F

Date: Mon, 30 Mar 2020 20:31:02 GMT

Server: Redacted

Content-Type: application/json;charset=UTF-8

Vary: Accept-Encoding

Strict-Transport-Security: max-age=63072000; includeSubDomains; preload


{

    "notFoundIds": null,

    "items": [

        {

            "id":

"Y2lzY29zcGFyazovL3VzL1BFT1BMRS9mZTg1ODhkMiliMjQ0LTQ2NmItYWU5ZS0xMDgwYjk0YzA0ZTk",

            "emails": [

                "networkdeveloper@thenetdev.com"

            ],

            "displayName": "Network Developer",

            "nickName": "networkdeveloper",

            "firstName": "networkdeveloper",

            "lastName": "networkdeveloper",

            "orgId": "Y2lzY29zcGFyazovL3VzL09SR0FOSVpBVElPTi9jb25zdW1lcg",

            "created": "2020-03-30T15:26:11.152Z",

            "lastActivity": "2020-03-30T20:28:29.221Z",

            "status": "active",

            "type": "person"

        }

    ]

}
```

The method, URI (including the email parameter), and Authentication header field are everything you need to perform this API call. The three objects are highlighted in the request in Example 17-32.

This API call is not entirely necessary because you may use the user's email in all subsequent requests. However, this call provides a good checkpoint to make sure the user you are trying to add to your team and space is actually registered on Webex Teams and that the first and last names are what you expected.

Now that you have retrieved the information for the user networkdeveloper, you need to create a team and add the user to it. To create a team, use the POST method and the URI https://api.ciscospark.com/v1/teams. The body of the request, at a minimum, should include the team name. For this request, you need to make sure that the Authorization header is also added. In addition, because this request contains a message body, the header Content-Type needs to be added, with the value application/json. This API call is documented under the Teams category. Example 17-33 shows both the HTTP request and response.

Example 17-33 *HTTP Request and Response for Creating a New Team*

```
! HTTP Request

POST /v1/teams HTTP/1.1

Authorization: Bearer MzUyYWI0ZjgtNzFmMC00OTgxLWJhMjctZWZhMWRjYjFjOWQ5NTA5YjczMDUtMDgz_PF84_consumer

Content-Type: application/json

User-Agent: PostmanRuntime/7.24.0

Accept: */*

Cache-Control: no-cache

Host: api.ciscospark.com

Accept-Encoding: gzip, deflate, br

Connection: keep-alive

Content-Length: 52

{

  "name": "Webex Teams Devs TEAM (via REST API)"

}
```

```
! HTTP Response

HTTP/1.1 200 OK

Via: 1.1 linkerd

Transfer-Encoding: chunked

Content-Encoding: gzip

TrackingID: ROUTER_5E830CAC-0AD9-01BB-00FB-5DA8916800FB

Date: Tue, 31 Mar 2020 09:26:12 GMT

Server: Redacted

Content-Type: application/json;charset=UTF-8

Vary: Accept-Encoding

Strict-Transport-Security: max-age=63072000; includeSubDomains; preload



{

    "id":

"Y2lzY29zcGFyazovL3VzL1RFQU0vYTliNjc0NDAtNzMzMS0xMWVhLWFjYjktZmZmOGM5NGY2ODVl",

    "name": "Webex Teams Devs TEAM (via REST API)",

    "creatorId":

"Y2lzY29zcGFyazovL3VzL1BFT1BMRS83ZjA0MTMzOC0xNGVhLTRiY2UtOTk0MS00MzUwZDgwZDRlYjM",

    "created": "2020-03-31T09:26:12.612Z"

}
```

The significant fields in the request are highlighted in Example 17-33. Notice the id field in the response, also highlighted in the example. The id value will be used in the next two API calls to identify the team that was just created.

Now to add user networkdeveloper to this newly created team, you need to use the POST method and the URI https://api.ciscospark.com/v1/team/memberships. This API call is documented under the category Team Memberships in the API reference. The body of the message, at a minimum, must include either personID or personEmail. Example 17-34 shows the easier option: the user's email. Make sure to include the same two headers as in Example 17-33.

Example 17-34 *The HTTP Request Body for Adding the User networkdeveloper to the Team Created in Example 17-33*

```
{

  "teamId":

"Y2lzY29zcGFyazovL3VzL1RFQU0vYTliNjc0NDAtNzMzMS0xMWVhLWFjYjktZmZmOGM5NGY2ODVl",

  "personEmail": "networkdeveloper@thenetdev.com"

}
```

The response body should look similar to the body in Example 17-35.

Example 17-35 *The HTTP Response Body Received for Making a Team Membership API Call*

```
{

    "id":

"Y2lzY29zcGFyazovL3VzL1RFQU1fTUVNQkVSU0hJUC9mZTg1ODhkMiliMjQ0LTQ2NmItYWU5ZS0xMDgwYjk0YzA

0ZTk6YTliNjc0NDAtNzMzMS0xMWVhLWFjYjktZmZmOGM5NGY2ODVl",

    "teamId":

"Y2lzY29zcGFyazovL3VzL1RFQU0vYTliNjc0NDAtNzMzMS0xMWVhLWFjYjktZmZmOGM5NGY2ODVl",

    "personId":

"Y2lzY29zcGFyazovL3VzL1BFT1BMRS9mZTg1ODhkMi1iMjQ0LTQ2NmItYWU5ZS0xMDgwYjk0YzA0ZTk",

    "personEmail": "networkdeveloper@thenetdev.com",

    "personDisplayName": "Network Developer",

    "personOrgId": "Y2lzY29zcGFyazovL3VzL09SR0FOSVpBVE1PTi9jb25zdWl1cg",

    "isModerator": false,

    "created": "2020-03-31T09:27:01.039Z"

}
```

The next step is to create a room attached to the team, and add the user networkdeveloper to that room. By navigating to the Rooms category and then to the Rooms Membership category in the API reference documentation, you should be able to figure out that you need to do two **POST** HTTP requests. The first, to create your room, uses the URI https://api.ciscospark.com/v1/rooms. To add the user to the room, you use the URI https://api.ciscospark.com/v1/memberships.

Keep in mind that a room is attached to a specific team, so you need the team ID that you received earlier, when creating the team. Then to add the user to the room, you need the room ID, which you should receive in the HTTP response after creating the room. Again, both requests should include the Authorization and Content-Type header fields.

When you complete all steps, the user interface of the Webex Teams application for the user networkdeveloper should be similar to Figure 17-

12.

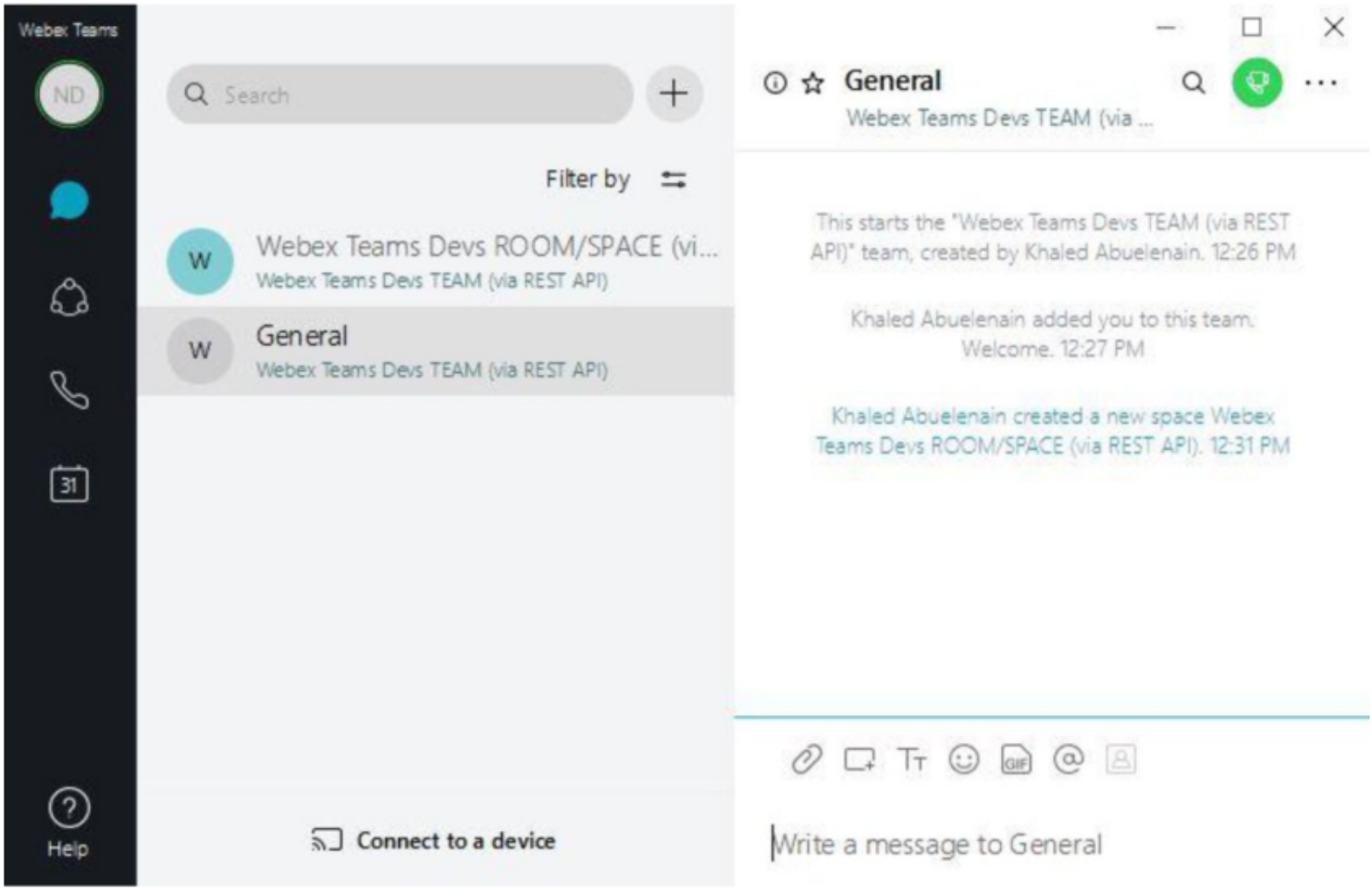

Figure 17-12 *The Webex Teams User Interface for the User networkdeveloper*

Finally, as documented under the Messages category, you need to perform an API call by using the POST method and the URI https://api.ciscospark.com/v1/messages and include both the Authorization and Content-Type header fields. The message body will, at a minimum, include the room ID and the message text that you need to send. Example 17-36 shows both the request and response.

Example 17-36 *The HTTP Request and Response for Sending a Message to a Specific Room*

```
! HTTP Request

POST /v1/messages HTTP/1.1

Authorization: Bearer

MzUyYWI0ZjgtNzFmMC00OTgxLWJhMjctZWZhMWRjYjFjOWQ5NTA5YjczMDUtMDgz_PF84_consumer

Content-Type: application/json

User-Agent: PostmanRuntime/7.24.0

Accept: */*

Cache-Control: no-cache

Host: api.ciscospark.com

Accept-Encoding: gzip, deflate, br

Connection: keep-alive

Content-Length: 184

{

  "roomId":

"Y2lzY29zcGFyazovL3VzLlJPT0OvNjc4ZjEyMTAtNzMzMi0xMWVhLWJlNjktMTU0M2E4ZjdhNDQ5",

  "text":"This is a test message from the Webex Teams REST API to the Webex Teams Devs"

}


! HTTP Response

HTTP/1.1 200 OK

Via: 1.1 linkerd

Transfer-Encoding: chunked

Content-Encoding: gzip

TrackingID: ROUTER_5E830EAF-0AEE-01BB-00D1-5DA8916800D1
```

```
Date: Tue, 31 Mar 2020 09:34:40 GMT
Server: Redacted
Content-Type: application/json;charset=UTF-8
Vary: Accept-Encoding
Strict-Transport-Security: max-age=63072000; includeSubDomains; preload

{
    "id":
"Y2lzY29zcGFyazovL3VzL01FU1NBR0UvZDhhMTRjNzAtNzMzMi0xMWVhLWEyZTQtMjk4ZmU0YzI0YWJm",
    "roomId":
"Y2lzY29zcGFyazovL3VzL1JPT00vNjc4ZjEyMTAtNzMzMi0xMWVhLWJlNjktMTU0M2E4ZjdhNDQ5",
    "roomType": "group",
    "text": "This is a test message from the Webex Teams REST API to the Webex Teams
Devs",
    "personId":
"Y2lzY29zcGFyazovL3VzL1BFT1BMRS83ZjA0MTMzOC0xNGVhLTRiY2UtOTk0MS00MzUwZDgwZDRlYjM",
    "personEmail": "kabuelenain@gmail.com",
    "created": "2020-03-31T09:34:40.823Z"
}
```

The message you sent now appears in the Webex Teams user interface, as shown in Figure 17-13.

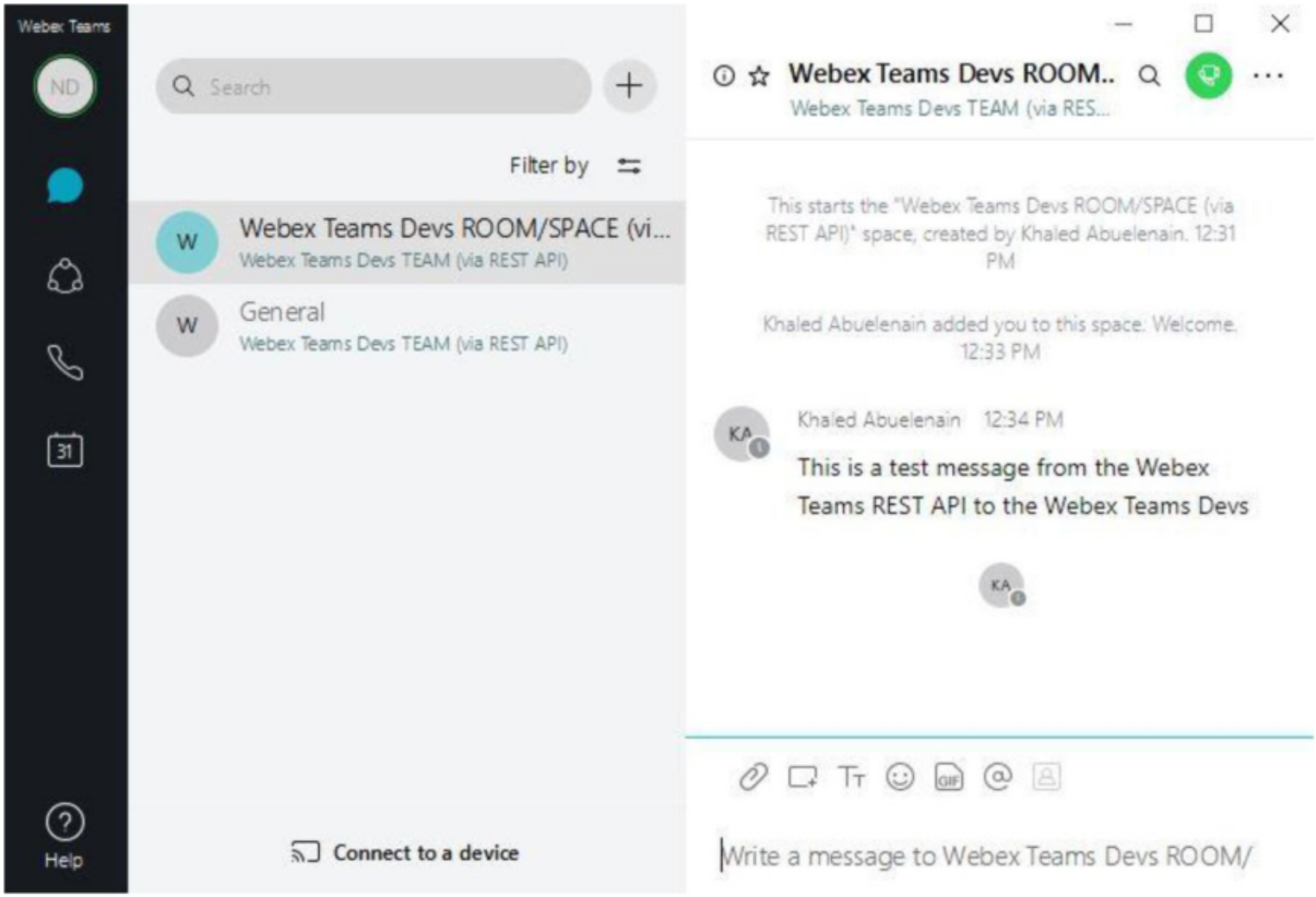

Figure 17-13 *The Teams User Interface, Showing the Message After the API Call*

Note

The user performing the API calls, including the last API call that sent a message to the newly created room, is Khaled Abuelenain. The Webex Teams account in Figures 17-12 and 17-13 belongs to the user networkdeveloper. This is why in Figure 17-12 you can see the message "Khaled Abuelenain added you to this team."

Summary

This chapter explores the programmability and automation features of several Cisco platforms. The following platforms are covered under their

respective domains: NX-OS, IOS XE, and IOS XR in the networks domain, Meraki in the cloud-managed networks domain, DNA Center in the network management systems and controllers domain, and the collaboration portfolio from Cisco (including an example involving Webex Teams).

The chapter presents use cases covering the following features:

• Linux shells on each of the three networking platforms

• NX-API CLI, NX-API REST and the Developer Sandbox on Open NX-OS

• NETCONF on each of the three networking platforms

• How to enable the Dashboard API on Meraki and use it to retrieve organizations, networks, and devices, as well as how to create a VLAN

• DNA Center and how to list sites and devices using its Intent API

• Cisco Webex Teams and how to create a team and a space/room, add people to the team and room, and send a message to a room using the Teams REST API

Using the information covered in this chapter, you can programmatically interface with virtually any Cisco platform using the APIs exposed by those platforms. All you need to do is understand the particulars of each API through that API's documentation, and you are good to go. Imagine the possibilities when coupling the information in this chapter with a programming language like Python that provides true automation capabilities.

The next chapter will cover the programmability and automation of a number of non-Cisco platforms.

Chapter 18. Programming Non-Cisco Devices

In Chapter 17, "Programming Cisco Devices," you learned about using the programmable interfaces on a variety of different platforms from Cisco, such as routers and switches running Cisco IOS XE, IOS XR, and NX-OS, as well Cisco DNA Center, Meraki, and Webex Teams. In this chapter, you will extend your knowledge to some non-Cisco network operating systems. This chapter covers some of the popular and emerging network operating systems to give you an overview of how to introduce programmability to network management practices with specific vendor platforms.

General Approaches to Programming Networks

As you have already learned in this book, you can use various interfaces to manage network platforms. Some of them are more popular or ubiquitous than others. The following sections provide an overview of several interfaces used at non-Cisco network operating systems.

The Vendor/API Matrix

When you hear about network programmability, you might be inclined to associate it with specific protocols, such as NETCONF, YANG, or HTTP/REST. However, network programmability is a much broader topic. As mentioned in Chapter 1, "The Network Programmability and Automation Ecosystem," programmability is the ability to manage platforms via *programmable interfaces* exposed by those platforms. Model-based programmability extends this paradigm to use data models. Network management involves tools and applications that add orchestration to the mix, and by using programmability, you can automate workflows involving several automated tasks performed in a specific sequence on, possibly, a number of different platforms. It's crucial that you keep this in mind and not limit yourself by unnecessary boundaries because you will run into programmability in places you never thought existed.

Programmability goes hand in hand with the different ways that network platforms can be managed. Figure 18-1 illustrates a sample API vendor mapping for a number of network operating systems. It includes Nokia SR OS and Juniper Junos OS, which, together with Cisco IOS XR, cover the majority of the service provider market in the United States, and Arista EOS and Cumulus Linux, which are very popular for big data centers and cloud providers.

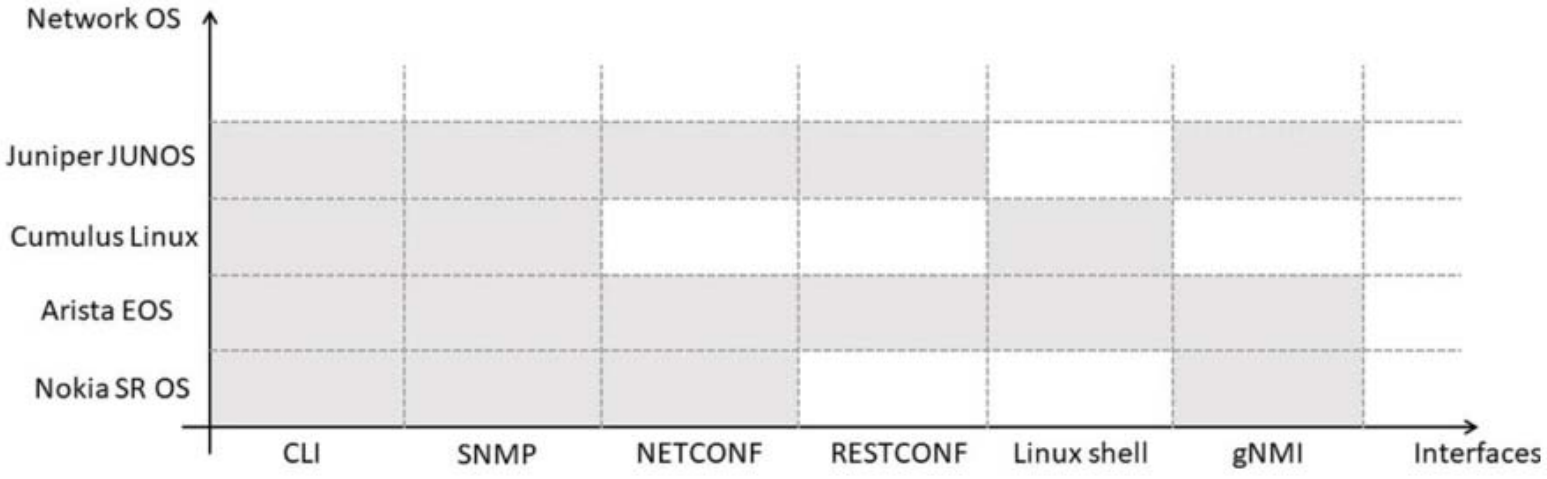

Figure 18-1 *Sample Vendor/API Mapping*

Figure 18-1 has two axes:

- The X-axis lists the types of interfaces, which allow us to interoperate with network elements.
- The Y-axis lists a subset of the vendors that exist on the market today.

If you operate a network built using a vendor not included in this figure, you should assess it based on the interfaces listed on the X-axis.

Programmability via the CLI

The most basic interface for interacting with network functions is the command-line interface (CLI). We cover the CLI under the umbrella of network programmability and automation because there are millions of legacy devices around the world that don't support NETCONF, RESTCONF, or any of the other APIs discussed so far in this book. Not being able to programmatically manage a device is certainly not a good enough reason to throw away the device if the device is performing its primary task (such routing and/or switching) well; companies do not swap these legacy devices with newer ones just to implement programmability and network automation. Therefore, the first step toward programmability and automation for such companies is to adapt their network management logic to manage legacy devices using the CLI alongside more modern devices using NETCONF, RESTCONF, gNMI, or other APIs.

When the Internet was still in its early days, finding trustworthy information on how to configure network elements was a challenging task. The number of competent network engineers was also considerably lower than it is today. The CLI was humorously referred to as the *cash line interface*. The jokes were justified, as knowledge of the CLI was a rare commodity. Today the situation is very different from the situation a couple decades ago. There are plenty of excellent resources today, including documentation covering detailed configuration for platforms from any vendor, various video trainings, and independent multivendor blogs. The newer humorous name for the CLI is *commodity line interface*. Today, the CLI is easy to understand and learn to use, and it still plays a significant role in network management.

From a software-oriented point of view, consider the following capabilities of CLI-based configuration management:

- The CLI configuration can be split into independent blocks.
- The blocks can be parametrized in terms of what is a variable and what is a fixed value.
- This configuration can be implemented in a network management system to convert the internal data modeling into the proper sequence of CLI commands.

Ansible provides a good example of CLI-based programmable network management, as you will see in Chapter 19, "Ansible." In addition, tools such as Cisco's NX-API CLI still involve using the CLI heavily to programmatically manage devices.

In short, in this era of programmability-based automation, knowledge of and ability to use the CLI are still very important.

Programmability via SNMP