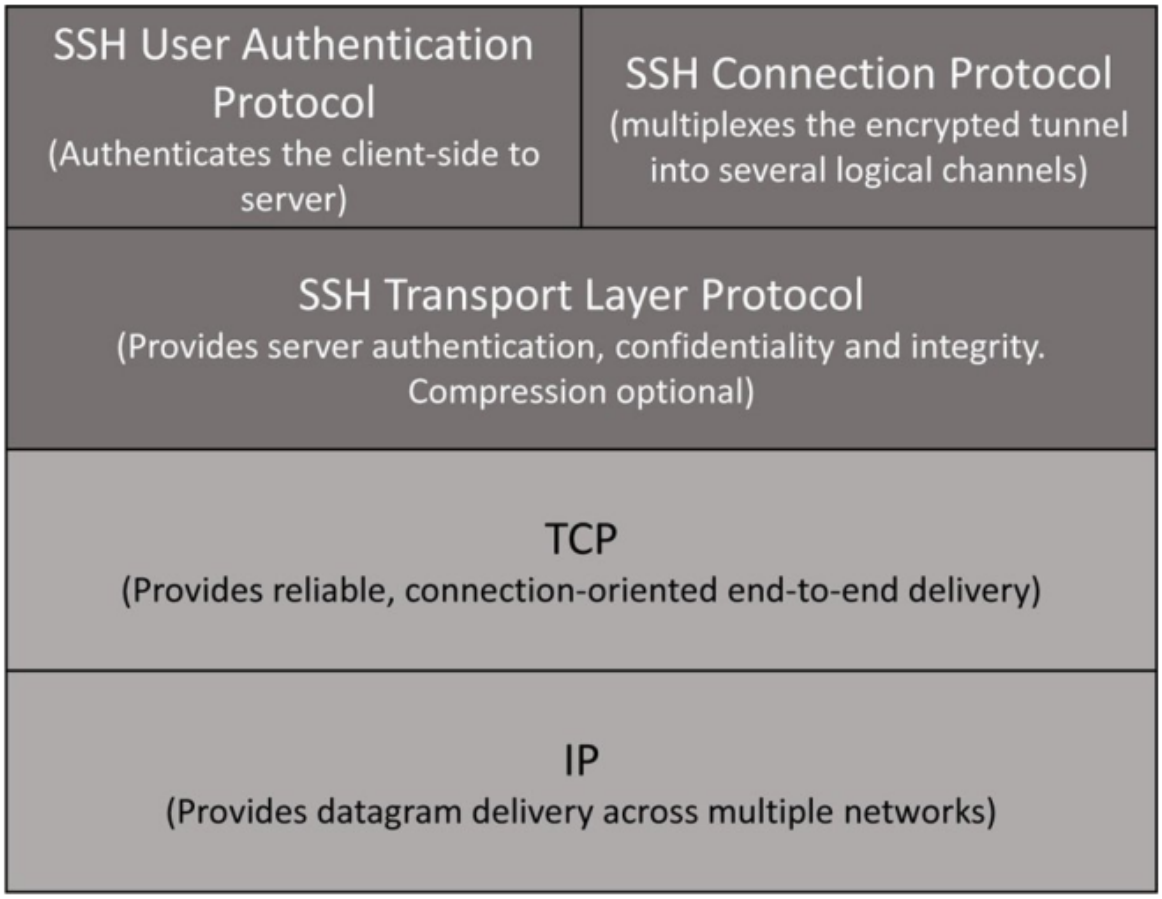

Figure 9-3 *SSH2 Layers*

SSH Transport Layer Protocol

The SSH Transport Layer Protocol (SSH-TRANS), as defined in RFC 4253, is a secure low-level transport protocol used to provide server authentication and initial key exchange and to set up strong encryption and integrity protection for the SSH protocol. The SSH-TRANS protocol provides host-based authentication and not user-based authentication. The following sequence of events happens with the SSH-TRANS protocol:

1. Connection setup: In this phase, the SSH connection established—usually on an IANA-registered TCP port (port 22). The connection works over a binary-transparent transport, and the underlying TCP transport protects against any transmission errors. Once the connection is established between the client and the server, they start exchanging data in the data portion of a TCP segment. The packet is exchanged in the following format:

• **Packet length**: This field indicates the length of the packet, in bytes.

• **Padding length**: This field indicates the length of random padding, in bytes.

• **Payload**: This field holds the useful contents of the packet.

• **Random padding**: This field indicates the arbitrary-length padding. The purpose of random padding is to ensure that the total length—including packet length, padding length, payload, and random padding field—is a multiple of the cipher block size or 8, whichever is greater.

• **MAC**: This field contains the Message Authentication Code (MAC) bytes if the message authentication has been negotiated; otherwise, it is set to 0 as initially the MAC algorithm is "none".

2. Protocol version exchange: In this step, the client sends a packet with an identification string in the following format:

```
SSH-protocolversion-softwareversion SP comments CR LF
```

where SP stands for space, CR stands for carriage return, and LF stands for line feed. The server responds to the client with its own identification string. During this process, the client and the server exchange SSH version and software information.

3. Algorithm negotiation: In this phase, each side sends an SSH_MSG_KEXINIT message that consists of a list of supported or allowed algorithms in order of preference from most to least. The algorithm negotiation also includes the exchange of packets for exchanging information on the encryption algorithm, MAC algorithm, key exchange, and the optional compression algorithm.

4. Diffie-Hellman key exchange: During this step, the key exchange happens via one of two Diffie-Hellman key exchange methods:

```
diffie-hellman-group1-sha1 REQUIRED

diffie-hellman-group14-sha1 REQUIRED
```

The Diffie-Hellman key exchange provides a shared secret that cannot be determined by either the client or the server alone. It is combined with the signature with the host key to provide host authentication. Note that RFC 4253 specifies support for alternate key exchange methods as well, but Diffie-Hellman is the method used most commonly. The output from the key exchange process is a shared key K and an exchange hash H. Both of these values are used for deriving encryption and authentication keys. The end of key exchange process is signaled by the exchange of SSH_MSG_NEWKEYS message, including the old keys and algorithms. All messages sent after this message use the new keys and algorithms.

5. Service request: The client sends an SSH_MSG_SERVICE_REQUEST message to request either the SSH Authentication Protocol or the SSH Connection Protocol.

Figure 9-4 illustrates the phases of SSH-TRANS protocol packet exchange.

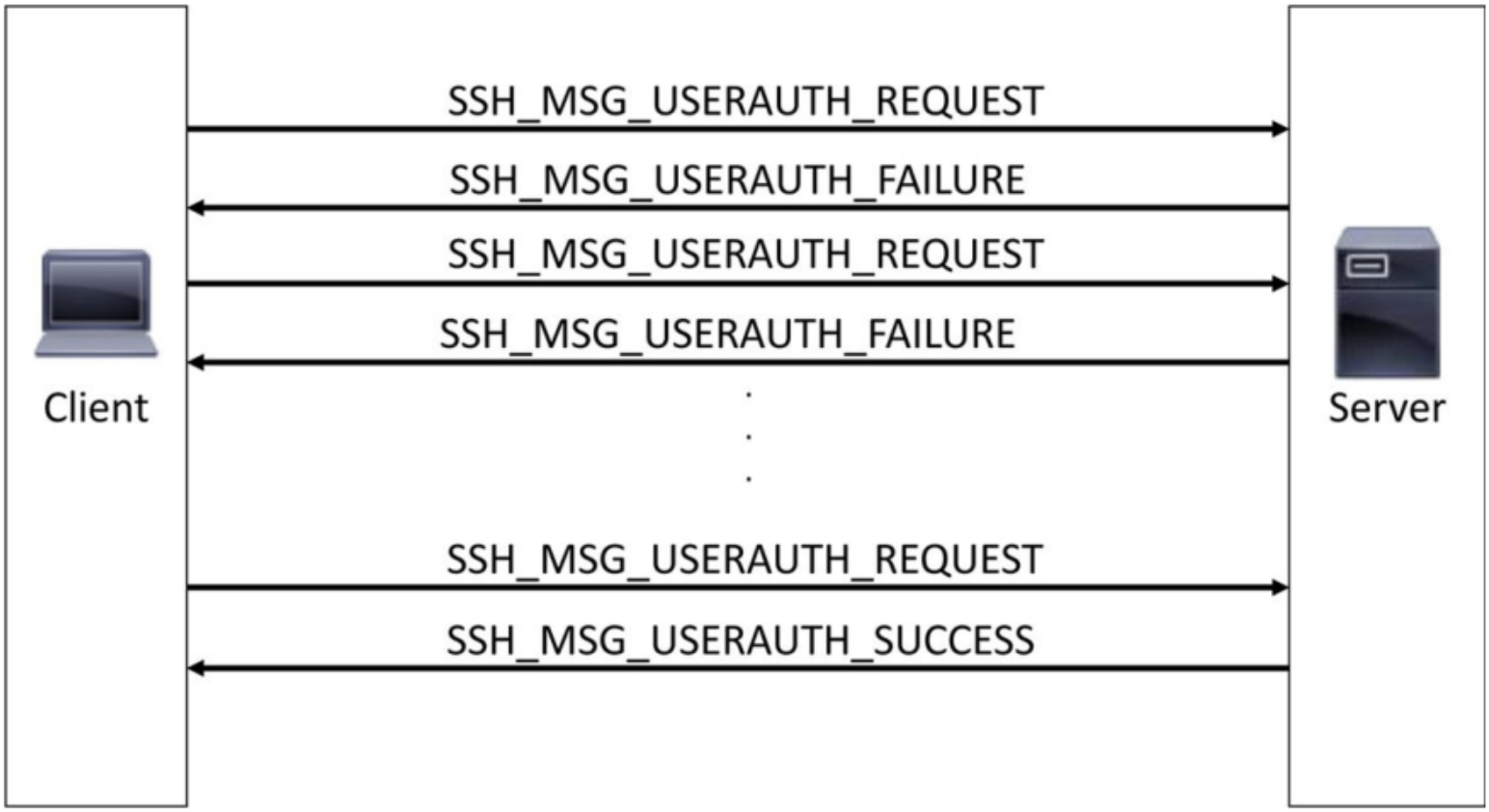

Figure 9-5 *SSH-AUTH Message Exchange*

As defined in RFC 4252, for the SSH Authentication Protocol, the server may require one or more of the following authentication methods:

• **Public key authentication (publickey)**: The **publickey** method depends on the chosen public key algorithm. The client sends to the server a message signed by client's private key. On receiving the message, the server checks whether the supplied key is acceptable for authentication. Once validated, the server validates for the correct signature. If both the key and the signature are validated, only then is the user authenticated; otherwise, the authentication is rejected. This is a required authentication method, and all SSH implementations of the SSH Authentication Protocol should support this method. The public key method is exchanged with the SSH_MSG_USERAUTH_REQUEST message in the following format:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name in ISO-10646 UTF-8 encoding
string    service name in US-ASCII
string    "publickey"
boolean   FALSE
string    public key algorithm name
string    public key blob
```

Note that the public key algorithms are defined in the SSH transport layer protocol specification, which is defined in RFC 4253. The public key blob field may contain certificates. The server responds to the request with either the SSH_MSG_USERAUTH_FAILURE message or the SSH_MSG_USERAUTH_PK_OK message, which is in the following format:

```
byte      SSH_MSG_USERAUTH_PK_OK
string    public key algorithm name from the request
string    public key blob from the request
```

Once the algorithm is negotiated, the client sends a signature that is generated using its private key. The signature is sent in the following packet format:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "publickey"
boolean   TRUE
string    public key algorithm name
string    public key to be used for authentication
string    signature
```

• **Password authentication (password)**: With this method, the client sends a message containing the password in plaintext. Even though the password is in plaintext, it is protected by the encryption from the SSH transport layer protocol. The password is sent with the SSH_MSG_USERAUTH_REQUEST packet in the following format:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
```

```
string    service name

string    "password"

boolean   FALSE

string    plaintext password in ISO-10646 UTF-8 encoding
```

The server responds with SSH_MSG_USERAUTH_SUCCESS if the authentication process is completed successfully or SSH_MSG_USERAUTH_FAILURE if the authentication fails or partially fails.

• **Host-based authentication (hostbased)**: With this method, the authentication is performed on the client's host rather than on the client itself. This is useful in scenarios where sites wish to allow authentication based on the host that the user is coming from and the username on the remote host. The client sends a signature created with the private key of the client host and is checked by the server with host's public key. When the identity is established for the client's host, authorization is performed based on the username on the server and the client and the client host name. The message for host-based authentication is sent in the following format:

```
byte      SSH_MSG_USERAUTH_REQUEST

string    user name

string    service name

string    "hostbased"

string    public key algorithm for host key

string    public host key and certificates for client host

string    client host name expressed as the FQDN in US-ASCII

string    user name on the client host - ISO-10646 UTF-8 encoding

string    signature
```

The server verifies the following:

• The host key actually belongs to the client host named in the message.

• The username provided is allowed to log in.

• The signature on the given host key is correct.

SSH Connection Protocol

The SSH Connection Protocol, as defined in RFC 4254, runs on top of the SSH-TRANS and SSH-USERAUTH protocols; that is, it runs on top of a secure connection provided by the SSH-TRANS protocol and assumes that a secure authenticated connection in use is provided by the SSH-USERAUTH protocol. The secure authentication connection is also referred to as a *tunnel*, and it is used by the connection protocol to multiplex a number of logical channels.

All types of communication, such as a terminal session or a forwarded connection, are supported using separate channels, which may be opened by either side. For each channel, each side associates a unique channel number, and that number may not be the same on both sides. Channels are flow controlled, which means that no data is sent to the channel until a message is received indicating that space is available for the TCP connection. The life of a channel progresses through three stages:

• Opening a channel

• Data transfer

• Closing a channel

When either side wishes to open a channel, it allocates a local number for the channel and sends a message in the following format:

```
byte      SSH_MSG_CHANNEL_OPEN

string    channel type in US-ASCII only

uint32    sender channel

uint32    initial window size

uint32    maximum packet size

....      channel type specific data follows
```

The message contains the local channel number and the initial window size. If the remote side is able to open the channel, it returns an SSH_MSG_CHANNEL_CONFIRMATION message that contains the sender channel number, window size, and packet size values of incoming traffic in the following format:

```
byte      SSH_MSG_CHANNEL_OPEN_CONFIRMATION

uint32    recipient channel

uint32    sender channel

uint32    initial window size

uint32    maximum packet size

....      channel type specific data follows
```

It is important to remember that the window size specifies how many bytes the remote side can send before it must wait for the window size to be adjusted by either side. The window size is adjusted by sending the following message:

```
byte      SSH_MSG_CHANNEL_WINDOW_ADJUST

uint32    recipient channel

uint32    bytes to add
```

If the remote side does not support the specified channel type, it simply responds with a SSH_MSG_CHANNEL_OPEN_FAILURE message in

the following format:

```
byte      SSH_MSG_CHANNEL_OPEN_FAILURE

uint32    recipient channel

uint32    reason code

string    description in ISO-10646 UTF-8 encoding [RFC3629]

string    language tag [RFC3066]
```

When the channel is open, data transfer is performed by sending an SSH_MSG_CHANNEL_DATA message that includes the recipient channel number and a block of data. The data transfer continues with the help of these messages as long as the connection is open. The channel data message is sent in the following format:

```
byte      SSH_MSG_CHANNEL_DATA

uint32    recipient channel

string    data
```

When either side wishes to terminate or close the channel, it sends an SSH_MSG_CHANNEL_CLOSE message that includes the recipient channel number.

RFC 4254 recognizes four types of channels for the SSH Connection Protocol:

- **session**
- **X11**
- **forwarded-tcpip**
- **direct-tcpip**

The channel type **session** is used for remote program execution. The program may be a shell or a file transfer application. When a session channel is opened, subsequent requests are used to start the remote program. To open a session, the following message is sent to the remote device:

```
byte      SSH_MSG_CHANNEL_OPEN

string    "session"

uint32    sender channel

uint32    initial window size

uint32    maximum packet size
```

The channel type **x11** refers to the channel for X Window System, which is a computer software system and network protocol that provides a graphical user interface (GUI) for networked computers. X Window System allows applications to run on a network server but displayed on a client desktop machine. For the **x11** channel, a message is sent in the following format:

```
byte      SSH_MSG_CHANNEL_OPEN

string    "x11"

uint32    sender channel

uint32    initial window size

uint32    maximum packet size

string    originator address (e.g., "192.168.0.100")

uint32    originator port
```

The channel types **forwarded-tcpip** and **direct-tcpip** are used for remote port forwarding and local port forwarding, respectively. To understand what these channel types do, you need to first understand what port forwarding means. In simple terms, port forwarding provides the ability to convert any insecure connection into a secure SSH connection; this is also sometimes referred to as *SSH tunneling*. Based on the port number, the incoming TCP traffic is delivered to the appropriate application. For instance, a system administrator can execute the command **ssh -L 80:demo.example.com:80 admin.example.com** to enable local port forwarding using SSH. The **-L** option with the **ssh** command is used for local port forwarding. For example, the command **ssh -L 80:demo.example.com:80 admin.example.com** opens a connection to the **admin.example.com** server and forwards any connection to port 80 on the local machine to port 80 on **demo.example.com**. Similarly, a system administrator can execute the command **ssh -R 8080:localhost:80 admin.example.com** to enable remote port forwarding. The **-R** option with the **ssh** command enables remote port forwarding. Remote port forwarding is useful in scenarios where a system administrator needs to provide access to the internal server. This command allows anyone on the remote server to connect to TCP port 8080 on the remote server. An SSH tunnel is established between the server and the client, and the client then establishes a TCP connection to port 80 on localhost.

With the **forwarded-tcpip** channel or remote forwarding, the user's SSH client acts on behalf of the server. The client, on receiving the traffic on a given destination port, forwards the traffic to the port that is mapped and sends it to the destination the user chooses. For the **forwarded-tcpip** channel, the message is sent in the following format:

```
byte      SSH_MSG_CHANNEL_OPEN

string    "forwarded-tcpip"

uint32    sender channel

uint32    initial window size

uint32    maximum packet size

string    address that was connected

uint32    port that was connected

string    originator IP address
```

```
uint32    originator port
```

With the **direct-tcpip** channel, or local forwarding, the client intercepts the selected application-level traffic based on the configured port mappings for the application and forwards it from an unsecured TCP connection to a secure SSH tunnel. The SSH server, on the other hand, sends the incoming traffic to the destination port dedicated to the client application. In the case of the **direct-tcpip** channel, the message is sent in the following format:

```
byte      SSH_MSG_CHANNEL_OPEN
string    "direct-tcpip"
uint32    sender channel
uint32    initial window size
uint32    maximum packet size
string    host to connect
uint32    port to connect
string    originator IP address
uint32    originator port
```

Note

Forwarded TCP/IP channels are independent of any sessions. That is, closing a channel may not indicate that the forwarded connection will be closed.

Setting Up SSH

Almost all web and application-hosting servers or server access should be enabled with secure access using SSH. SSH is supported on all Linux/UNIX and Windows operating systems. Both Linux/UNIX and Windows-based systems support OpenSSH. When deploying SSH, users and system administrators can opt for either password-based SSH authentication or SSH key-based authentication, which involves using public and private keys to authenticate a user. The following sections look at how to deploy SSH on a CentOS 7 server with both password and SSH key-based authentication.

Setting Up SSH on CentOS

SSH can provide secure access to a server using various methods. One of the methods most commonly used by an SSH server is password-based authentication. This is an easy-to-use method, but it is not the most secure method because the passwords are usually not long enough to be resistant to brute-force attacks. Attackers can apply brute-force techniques to break the password and get access to the servers. In addition, applications, such as fail2ban provide additional security, but SSH keys are more reliable and provide more security for SSH authentication. When installing OpenSSH, password authentication is enabled by default.

Most UNIX/Linux-based operating systems have the OpenSSH package installed by default. However, if you need to manually install OpenSSH, you can use the **yum** package manager on CentOS to install the package, as shown in Example 9-1.When the package is installed, you can use the **systemctl** commands to start and enable the **sshd** process.

Example 9-1 *Installing OpenSSH on CentOS*

```
! Using yum package installer to install openssh-server and openssh-clients
# yum -y install openssh-server openssh-clients


! Using systemctl command to start sshd service
# systemctl start sshd


! Using systemctl command to automatically start sshd service at bootup
# systemctl enable sshd
```

By default, most SSH installations have password authentication enabled. This authentication method does not require any additional configuration but is not very secure. If password authentication is disabled, you need to edit the /etc/ssh/sshd_config file so that the **PasswordAuthentication** value is set to **yes**. When this value is set, you can restart the **sshd** process by using the **systemctl** command. This allows you to use password authentication for SSH login. Example 9-2 shows how to enable password authentication and perform SSH login by using this method. Notice that once the remote system has been successfully authenticated, the host is permanently added to the known hosts list in the user's .ssh/known_hosts file.

Example 9-2 *SSH Using Password Authentication*

```
[root@centos2 ~]# vi /etc/ssh/sshd_config
. . .


# To disable tunneled clear text passwords, change to no here!
#PasswordAuthentication yes
#PermitEmptyPasswords no


PasswordAuthentication yes
. . .
```

```
[root@centos2 ~]# systemctl restart sshd.service
```

```
[root@centos1 ~]# ssh root@10.1.101.101
The authenticity of host '10.1.101.101 (10.1.101.101)' can't be established.
ECDSA key fingerprint is SHA256:R5JwGwo4S844s7y9PKyZLDLKi3NfxGr0FK6Psa2YuKk.
ECDSA key fingerprint is MD5:e1:18:f6:0d:a1:7b:5e:f0:3a:76:6b:e8:23:c1:d6:49.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.1.101.101' (ECDSA) to the list of known hosts.
root@10.1.101.101's password:
Last login: Sun Oct 20 04:03:45 2019
[root@centos2 ~]#
[root@centos2 ~]# exit
logout
Connection to 10.1.101.101 closed.
```

```
[root@centos1 ~]# cd .ssh/
[root@centos1 .ssh]# cat known_hosts
10.1.100.1 ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCSLMuhVlsDiLV4azfbG9SPqtjWBmna09htUJ553j/PrLVkebIss
58ljmFHAOZ42vPU4FyLfVmmWtPkFjdljELGTwNtA7caYRJ6LlcbD68pEV3+222D1sNQpq/ddFl6vcRsRI
zEl5S5BIinLEjYnQEzwHNt9RrmvJHnlHPPL83YBBjBIsQrGz5hEWw72DJ/mvvlbo5eGMnvWav2wJSZHis
ug/6EKgUzdRy8tuzeTk61aJgedYCaP5QgElZkin2CWnjou/GyiUsFCy1HCfYGmzUqtkKnmYRmuYnIsAut
hH6pP8TdnnFteOlv6UVmcqusTBJAWPLgcP9hWdwkWIL894Jf
10.1.101.101 ecdsa-sha2-nistp256
AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBFkcnWu7aiEGrC0LxNQfW/QTkRZEi
zhOItpuihCuKM3xv3BqZ6zCSUw9RHOWH2TdDKqafV1r7jfAgh72Tu3aIdQ=
[root@centos1 .ssh]#
```

As mentioned earlier, another, more secure authentication method can be used instead of password authentication. SSH key-based authentication is a more secure method of authentication as it relies on public and private keys. To enable SSH key-based authentication, the first step is to generate an SSH key on the client machine. This can be done by using the command **ssh-keygen**. The SSH key generator (**ssh-keygen**) tool, which comes with the SSH package, generates an RSA- or DSA-based key that is used for public key authentication. If an existing SSH RSA/DSA key is already present, the tool gives you an option to override the existing key. It is important to note that RSA keys are recommended over DSA keys because DSA keys are 1024 bits only, whereas RSA keys can go up to 4096 bits. By default, the **ssh-keygen** tool generates a 2048-bit RSA key as shown in Example 9-3.

Example 9-3 *Generating an SSH Key*

```
[root@centos1 ~]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
/root/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:b7jsrGEORgCyixvS8aK2hC7nLc7+Kw2LScX1rE+gW40 root@centos1.cisco.com
The key's randomart image is:
+---[RSA 2048]----+
|o                |
|.o   .           |
|. + . o          |
|.o * . o         |
|= + + = S        |
|o=.+ E o o       |
|== += = . o      |
|B.*+.+ = o       |
|.B=++.oo=        |
+----[SHA256]-----+
```

Note

To implement a more secure key, it is recommended to use a larger key size—that is, 4096 bits. You can do this by using the following command:

```
ssh-keygen -b 4096
```

You can choose between using an RSA key or a DSA key by using the **-t** option with either the **rsa** or **dsa** keyword. By default, **ssh-keygen** generates an RSA key.

The **ssh-keygen** tool generates both a public key and a private key on the client machine, in the .ssh folder inside user's home directory. The public key should be copied onto the server. This can be done using the **ssh-copy-id** tool, which is installed along with the SSH package, along with *user@remote-server-address*. This tool performs an SSH login using the password authentication method and copies the public key onto the server. Example 9-4 demonstrates how the **ssh-copy-id** tool is used to copy the public key from the client machine onto the remote server.

Example 9-4 *Copying a Client Public Key onto a Server*

```
[root@centos1 ~]# cd .ssh
[root@centos1 .ssh]# pwd
/root/.ssh
[root@centos1 .ssh]# ls
authorized_keys  id_rsa  id_rsa.pub  known_hosts


[root@centos1 .ssh]# cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC9NBGVjxOqYI6yXJBf9fRqevxYOn4JuWmOUAGmYJxodC+Fs7OLd
h5E/8L83q3OOJSEEDDVDtsB88vMtT3vvMB+efvhzHWaTn4fEyE56AKED53sz+vU3V4Xk8n0sfBVYEbtpz
fDvmP+IIYFX2NN15AL6DQGay3V1t9fES2hRsmswBjrHgznlHdY/9LknzigIQWY7pQ/cdmwLGBjzCp6fk+
8y6M5evjZleobGLaGAbOmT7nbgJZ3jdGjqjln9J8fyzc+7aszxIquJIAVW+CAEv7L4huY8c5gPUljS+GY
kzwpKuNnWp5W6FloIi0RKxkVywLX4LFzRXwou6hr3KKxcC1f root@centos1.cisco.com
[root@centos1 .ssh]#
```

```
[root@centos1 ~]# ssh-copy-id root@10.1.101.101
/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/root/.ssh/id_rsa.pub"
/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out
any that are already installed
/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted
now it is to install the new keys
root@10.1.101.101's password:


Number of key(s) added: 1


Now try logging into the machine, with:   "ssh 'root@10.1.101.101'"
and check to make sure that only the key(s) you wanted were added.
 [root@centos1 ~]#
```

When the public key has been copied on the server, the user can establish an SSH session from the client machine to the server by using the SSH keys. The public key from the client machine is copied to the authorized_keys file on the server under the .ssh directory. Example 9-5 shows an SSH session established using SSH key-based authentication on the server.

Example 9-5 *Establishing an SSH Session Using SSH Key-Based Authentication*

```
[root@centos1 ~]# ssh root@10.1.101.101
Enter passphrase for key '/root/.ssh/id_rsa':
Last login: Sun Oct 20 05:54:07 2019 from 10.1.100.100
[root@centos2 ~]# cd .ssh
[root@centos2 .ssh]# ls
authorized_keys  id_rsa  id_rsa.pub  known_hosts
[root@centos2 .ssh]# cat authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC9NBGVjxOqYI6yXJBf9fRqevxYOn4JuWmOUAGmYJxodC+Fs7OLd
h5E/8L83q3OOJSEEDDVDtsB88vMtT3vvMB+efvhzHWaTn4fEyE56AKED53sz+vU3V4Xk8n0sfBVYEbtpz
fDvmP+IIYFX2NN15AL6DQGay3V1t9fES2hRsmswBjrHgznlHdY/9LknzigIQWY7pQ/cdmwLGBjzCp6fk+
8y6M5evjZleobGLaGAbOmT7nbgJZ3jdGjqjln9J8fyzc+7aszxIquJIAVW+CAEv7L4huY8c5gPUljS+GY
kzwpKuNnWp5W6FloIi0RKxkVywLX4LFzRXwou6hr3KKxcC1f root@centos1.cisco.com
[root@centos2 .ssh]#
```

Even after enabling SSH key-based authentication, password-based authentication is still active and should be disabled manually. However, password-based authentication should not be disabled before the SSH key-based authentication is successfully set up for the user account. You can disable password authentication for SSH by editing the sshd_config file in the /etc/ssh/ directory. Within this file, change the setting for **PasswordAuthentication** to **no**, as shown in Example 9-6. After editing the file, save and exit the file by pressing Esc+x and then Enter. After the file has been modified and saved, you need to restart the **sshd** service in order for the changes to take effect.

Example 9-6 *Disabling Password-Based Authentication*

```
[root@centos-1-srvr ~]# vi /etc/ssh/sshd_config

. . .


# To disable tunneled clear text passwords, change to no here!

#PasswordAuthentication yes

#PermitEmptyPasswords no


PasswordAuthentication no

. . .


[root@centos-1-srvr ~]# systemctl restart sshd.service
```

Enabling SSH on Cisco Devices

Secure terminal sessions are necessary not just for servers and hosts but also for network devices. Cisco devices support both Telnet and SSH-based login for performing network configuration via terminal sessions or performing validation of the output from various routing protocols. Different Cisco network operating systems have different features enabled by default. For instance, Cisco IOS/IOS XE and Cisco IOS XR software does not have SSH enabled by default, but NX-OS software does. It is recommended to use secure methods and limit login access to network devices to specific IP addresses or specific users. The following sections examine how SSH-based login can be enabled with added security configured on all three of these Cisco network operating systems.

Configuring and Verifying SSH on Cisco IOS XE

Cisco IOS XE supports both SSH1 and SSH2. You use the command **ip ssh version** [**1** | **2**] to specify which SSH version the device will be running. Because SSH1 is not standardized and is comparatively less secure than SSH2, it is recommended to run SSH2. If the command **ip ssh version** is not configured, the IOS XE device runs in default compatibility mode, which means both SSH1 and SSH2 connections will be accepted by the device.

SSH2 has been enhanced to add more capabilities to a device. Some of these capabilities include virtual routing and forwarding (VRF)-aware SSH and Diffie-Hellman (DH) group exchange support. With VRF-aware SSH, you can now use SSH to access devices on the IP addresses that are part of either a management or non-default VRF instance. Traditionally, Cisco IOS XE implementations supported 768-bit modulus for performing encryption. With the increasing need for tightened security capabilities, support for larger key sizes to accommodate DH group 14 (2048 bits) and DH group 16 (4096 bits) cryptographic applications between server and client has become necessary. Remember that the DH groups determine the strength of the key used in the key exchange process. Additional time is required to compute the key for higher group numbers as they are more secure. The command **ip ssh dh min size** can be used to configure the modulus size on an SSH server.

There are two way to enable SSH2 on Cisco IOS XE devices. One way is to assign a hostname and a domain name to a device and generate an RSA key. The other way is to use RSA key pairs instead of configuring domain names. In this method, RSA keys are generated with key labels and modulus sizes using the command **crypto key generate rsa usage-keys label** *key-label* **modulus** *modulus-size* and, then the *key-label* value is assigned to the command **ip ssh rsa keypair-name** *keypair-name*. Example 9-7 illustrates the configuration of SSH using both of these methods on a Cisco IOS XE device.

Example 9-7 *Configuring SSH2 on Cisco IOS XE*

```
! SSH Version 2 using hostname and domain name


router(config)#ip ssh version 2

Please create RSA keys to enable SSH (and of atleast 768 bits for SSH v2).


router(config)#hostname R2-A903-XE

R2-A903-XE(config)#ip domain name cisco.com

R2-A903-XE(config)#username cisco privilege 15 password cisco

R2-A903-XE(config)#crypto key generate rsa

The name for the keys will be: R2-A903-XE.cisco.com

Choose the size of the key modulus in the range of 360 to 4096 for your

  General Purpose Keys. Choosing a key modulus greater than 512 may take

  a few minutes.


How many bits in the modulus [512]: 768

% Generating 768 bit RSA keys, keys will be non-exportable...

[OK] (elapsed time was 6 seconds)
```

```
R2-A903-XE(config)#ip ssh version 2
```

```
! SSH Version 2 using RSA key pairs


Router(config)#host R1-A903-XE
R1-A903-XE(config)#username cisco privilege 15 password cisco
R1-A903-XE(config)#crypto key generate rsa usage-keys label sshkeys modulus 2048
The name for the keys will be: sshkeys


% The key modulus size is 2048 bits
% Generating 2048 bit RSA keys, keys will be non-exportable...
[OK] (elapsed time was 8 seconds)
% Generating 2048 bit RSA keys, keys will be non-exportable...
[OK] (elapsed time was 3 seconds)


R1-A903-XE(config)#ip ssh rsa keypair-name sshkeys
R1-A903-XE(config)#ip ssh version 2
```

When an external client tries to connect the IOS XE device, the complete SSH packet exchange can either be viewed using a SPAN session or by running the **debug ip ssh** command on the device. This **debug** command displays all the packet exchange related to SSH, including various SSH messages and DF algorithms. Example 9-8 illustrates a client connecting to an IOS XE device using SSH. Notice that because the RSA keys were generated with modulus 2048, DF group 14 is being selected as the key exchange algorithm.

Example 9-8 *SSH Debugging*

```
! Client on MAC connecting Cisco Device
Centos-srvr:~ root$ ssh cisco@172.16.223.26
Password:


R1-A903-XE#
```

```
R1-A903-XE#debug ip ssh
*Mar  4 03:21:26.529: SSH0: starting SSH control process
*Mar  4 03:21:26.529: SSH0: sent protocol version id SSH-2.0-Cisco-1.25
*Mar  4 03:21:26.676: SSH0: protocol version id is - SSH-2.0-OpenSSH_7.9
*Mar  4 03:21:26.676: SSH2 0: kexinit sent: kex algo = diffie-hellman-group-
exchange-sha1,diffie-hellman-group14-sha1
*Mar  4 03:21:26.676: SSH2 0: Server certificate trustpoint not found. Skipping
hostkey algo = x509v3-ssh-rsa
*Mar  4 03:21:26.676: SSH2 0: kexinit sent: hostkey algo = ssh-rsa
*Mar  4 03:21:26.676: SSH2 0: kexinit sent: encryption algo = aes128-ctr,aes192-
ctr,aes256-ctr
*Mar  4 03:21:26.676: SSH2 0: kexinit sent: mac algo = hmac-sha2-256,hmac-sha2-
512,hmac-sha1,hmac-sha1-96
*Mar  4 03:21:26.676: SSH2 0: send:packet of  length 312 (length also includes
padlen of 4)
*Mar  4 03:21:26.677: SSH2 0: SSH2_MSG_KEXINIT sent
*Mar  4 03:21:26.677: SSH2 0: ssh_receive: 536 bytes received
*Mar  4 03:21:26.679: SSH2 0: input: total packet length of 1392 bytes
*Mar  4 03:21:26.679: SSH2 0: partial packet length(block size)8 bytes,needed
1384 bytes,
               maclen 0
*Mar  4 03:21:26.679: SSH2 0: ssh_receive: 536 bytes received
*Mar  4 03:21:26.679: SSH2 0: partial packet length(block size)8 bytes,needed
1384 bytes,
               maclen 0
R1-A903-XE#
*Mar  4 03:21:26.680: SSH2 0: ssh_receive: 320 bytes received
*Mar  4 03:21:26.680: SSH2 0: partial packet length(block size)8 bytes,needed
```

```
1384 bytes,
               maclen 0
*Mar  4 03:21:26.680: SSH2 0: input: padlength 4 bytes
*Mar  4 03:21:26.680: SSH2 0: SSH2_MSG_KEXINIT received
*Mar  4 03:21:26.680: SSH2 0: kex: client->server enc:aes128-ctr mac:hmac-sha2-
256
*Mar  4 03:21:26.680: SSH2 0: kex: server->client enc:aes128-ctr mac:hmac-sha2-
256
*Mar  4 03:21:26.681: SSH2 0: Using kex_algo = diffie-hellman-group14-sha1
*Mar  4 03:21:26.749: SSH2 0: expecting SSH2_MSG_KEXDH_INIT
*Mar  4 03:21:26.819: SSH2 0: ssh_receive: 272 bytes received
*Mar  4 03:21:26.819: SSH2 0: input: total packet length of 272 bytes
*Mar  4 03:21:26.820: SSH2 0: partial packet length(block size)8 bytes,needed 264
bytes,
               maclen 0
*Mar  4 03:21:26.820: SSH2 0: input: padlength 6 bytes
*Mar  4 03:21:26.820: SSH2 0: SSH2_MSG_KEXDH_INIT received
*Mar  4 03:21:26.988: SSH2 0: signature length 271
*Mar  4 03:21:26.988: SSH2 0: send:packet of  length 832 (length also includes
padlen of 7)
*Mar  4 03:21:26.988: SSH2: kex_derive_keys complete
*Mar  4 03:21:26.989: SSH2 0: send:packet of  length 16 (length also includes padlen of 10)
*Mar  4 03:21:26.989: SSH2 0: newkeys: mode 1
*Mar  4 03:21:26.989: SSH0: TCP send failed enqueueing
*Mar  4 03:21:27.139: SSH2 0: SSH2_MSG_NEWKEYS sent
*Mar  4 03:21:27.139: SSH2 0: waiting for SSH2_MSG_NEWKEYS
*Mar  4 03:21:27.139: SSH2 0: ssh_receive: 16 bytes received
*Mar  4 03:21:27.139: SSH2 0: input: total packet length of 16 bytes
*Mar  4 03:21:27.139: SSH2 0: partial packet length(block size)8 bytes,needed 8
bytes,
               maclen 0
*Mar  4 03:21:27.139: SSH2 0: input: padlength 10 bytes
*Mar  4 03:21:27.139: SSH2 0: newkeys: mode 0
*Mar  4 03:21:27.139: SSH2 0: SSH2_MSG_NEWKEYS received
*Mar  4 03:21:27.279: SSH2 0: ssh_receive: 64 bytes received
*Mar  4 03:21:27.279: SSH2 0: input: total packet length of 32 bytes
*Mar  4 03:21:27.279: SSH2 0: partial packet length(block size)16 bytes,needed
16 bytes,
               maclen 32
*Mar  4 03:21:27.279: SSH2 0: MAC compared for #3 :ok
*Mar  4 03:21:27.279: SSH2 0: input: padlength 10 bytes
*Mar  4 03:21:27.279: SSH2 0: send:packet of  length 32 (length also includes
padlen of 10)
*Mar  4 03:21:27.279: SSH2 0: computed MAC for sequence no.#3 type 6
*Mar  4 03:21:27.280: SSH2 0: Authentications that can continue =
publickey,keyboard-interactive,password
*Mar  4 03:21:27.421: SSH2 0: ssh_receive: 80 bytes received
*Mar  4 03:21:27.421: SSH2 0: input: total packet length of 48 bytes
*Mar  4 03:21:27.421: SSH2 0: partial packet length(block size)16 bytes,needed 32
bytes,
               maclen 32
*Mar  4 03:21:27.421: SSH2 0: MAC compared for #4 :ok
*Mar  4 03:21:27.421: SSH2 0: input: padlength 7 bytes
```

```
*Mar  4 03:21:27.421: SSH2 0: Using method = none
*Mar  4 03:21:27.421: SSH2 0: Authentications that can continue =
publickey,keyboard-interactive,password
*Mar  4 03:21:27.421: SSH2 0: send:packet of  length 64 (length also includes
padlen of 14)
*Mar  4 03:21:27.422: SSH2 0: computed MAC for sequence no.#4 type 51
*Mar  4 03:21:27.563: SSH2 0: ssh_receive: 112 bytes received
R1-A903-XE#
*Mar  4 03:21:27.563: SSH2 0: input: total packet length of 80 bytes
*Mar  4 03:21:27.563: SSH2 0: partial packet length(block size)16 bytes,needed 64
bytes,
               maclen 32
*Mar  4 03:21:27.563: SSH2 0: MAC compared for #5 :ok
*Mar  4 03:21:27.563: SSH2 0: input: padlength 15 bytes
*Mar  4 03:21:27.563: SSH2 0: Using method = keyboard-interactive
*Mar  4 03:21:27.564: SSH2 0: send:packet of  length 48 (length also includes
padlen of 11)
*Mar  4 03:21:27.564: SSH2 0: computed MAC for sequence no.#5 type 60
R1-A903-XE#
*Mar  4 03:21:29.999: SSH2 0: ssh_receive: 96 bytes received
*Mar  4 03:21:29.999: SSH2 0: input: total packet length of 64 bytes
*Mar  4 03:21:29.999: SSH2 0: partial packet length(block size)16 bytes,needed 48 bytes,
               maclen 32
*Mar  4 03:21:29.999: SSH2 0: MAC compared for #6 :ok
*Mar  4 03:21:29.999: SSH2 0: input: padlength 45 bytes
*Mar  4 03:21:30.000: SSH2 0: send:packet of  length 16 (length also includes
padlen of 10)
*Mar  4 03:21:30.000: SSH2 0: computed MAC for sequence no.#6 type 52
*Mar  4 03:21:30.000: SSH2 0: authentication successful for cisco
*Mar  4 03:21:30.145: SSH2 0: ssh_receive: 80 bytes received
! Output omitted for brevity
```

A Cisco IOS XE device can also act as an SSH client and can be used to connect to any remote device. In Example 9-9, the device R1-A903-XE is acting as a client to connect to a host named R2-A903-XE. Because on R2-A903-XE, the RSA key has been generated with modulus 768, DF group 1 is being used, as shown in Example 9-9.

Example 9-9 *A Cisco Device as SSH Client and SSH Debugging*

```
R1-A903-XE#ssh -l cisco 10.1.2.2
Password:




R2-A903-XE#
```

```
R2-A903-XE#debug ip ssh
Incoming SSH debugging is on


*Oct  8 07:54:29.525: %SYS-5-CONFIG_I: Configured from console by console
*Oct  8 07:54:31.724: SSH0: starting SSH control process
*Oct  8 07:54:31.724: SSH0: sent protocol version id SSH-2.0-Cisco-1.25
*Oct  8 07:54:31.727: SSH0: protocol version id is - SSH-2.0-Cisco-1.25
*Oct  8 07:54:31.727: SSH2 0: kexinit sent: kex algo = diffie-hellman-group-
exchange-sha1,diffie-hellman-group14-sha1
*Oct  8 07:54:31.727: SSH2 0: Server certificate trustpoint not found. Skipping
hostkey algo = x509v3-ssh-rsa
*Oct  8 07:54:31.727: SSH2 0: kexinit sent: hostkey algo = ssh-rsa
*Oct  8 07:54:31.727: SSH2 0: kexinit sent: encryption algo = aes128-ctr,aes192-
ctr,aes256-ctr
```

```
*Oct  8 07:54:31.727: SSH2 0: kexinit sent: mac algo = hmac-sha2-256,hmac-sha2-
512,hmac-sha1,hmac-sha1-96
*Oct  8 07:54:31.727: SSH2 0: send:packet of  length 312 (length also includes
padlen of 4)
*Oct  8 07:54:31.727: SSH2 0: SSH2_MSG_KEXINIT sent
*Oct  8 07:54:31.728: SSH2 0: ssh_receive: 312 bytes received
*Oct  8 07:54:31.728: SSH2 0: input: total packet length of 312 bytes
*Oct  8 07:54:31.728: SSH2 0: partial packet length(block size)8 bytes,needed 304
bytes,
               maclen 0
*Oct  8 07:54:31.728: SSH2 0: input: padlength 4 bytes
*Oct  8 07:54:31.728: SSH2 0: SSH2_MSG_KEXINIT received
*Oct  8 07:54:31.728: SSH2 0: kex: client->server enc:aes128-ctr mac:hmac-sha2-
256
*Oct  8 07:54:31.729: SSH2 0: kex: server->client enc:aes128-ctr mac:hmac-sha2-
256
*Oct  8 07:54:31.729: SSH2 0: Using kex_algo = diffie-hellman-group-exchange-sha1
*Oct  8 07:54:31.730: SSH2 0: ssh_receive: 24 bytes received
*Oct  8 07:54:31.730: SSH2 0: input: total packet length of 24 bytes
! Output omitted for brevity
```

Note

It is also possible to connect to Cisco devices by using the authentication method specified in AAA configuration. AAA configuration is beyond the scope of this chapter. Refer to the Cisco documentation for AAA configuration on the appropriate Cisco platforms and software.

Configuring SSH on IOS XR

SSH on Cisco IOS XR routers is not enabled by default. The Cisco IOS XR supports SSH1 with RSA-based keys and SSH2 with both RSA and Digital Signature Algorithm (DSA). To enable SSH, the IOS XR router should have the **k9sec** package installed or have a full **k9** image installed on the system; you can verify the presence of this image by using the command **show install active summary**, as shown in Example 9-10.

Example 9-10 ***show install active summary*** *Command Output*

```
RP/0/0/CPU0:XR-2#show install active summary
Sun Oct 13 04:38:23.106 UTC
Default Profile:
  SDRs:
    Owner
  Active Packages:
    disk0:xrvr-fullk9-x-6.1.2
```

Because it is recommended to use SSH2, you should generate an RSA or DSA key. You can do this by using the command **crypto key generate** [**rsa** | **dsa**]. This command takes in the modulus values with which the RSA or DSA key can be generated. The next step is to enable the SSH server on the IOS XR device. You can do this by using the configuration command **ssh server** [**vrf** *vrf-name*] **v2** in global configuration mode. The **vrf** option allows SSH sessions to be established to an IP address within the VRF instance specified. With just these two commands, you should be able to use SSH to access the IOS XR device, as shown in Example 9-11.

Example 9-11 *Enabling SSH2 on IOS XR*

```
RP/0/0/CPU0:XR-2#crypto key generate rsa
Sun Oct 13 04:39:54.010 UTC
The name for the keys will be: the_default
  Choose the size of the key modulus in the range of 512 to 4096 for your General
Purpose Keypair. Choosing a key modulus greater than 512 may take a few minutes.

How many bits in the modulus [2048]:
Generating RSA keys ...
Done w/ crypto generate keypair
[OK]
RP/0/0/CPU0:XR-2(config)#ssh server ?
  dscp           Cisco ssh server DSCP
```

```
  ipv4           IPv4 access list for ssh server
  ipv6           IPv6 access list for ssh server
  logging        Enable ssh server logging
  netconf        start ssh service for netconf subsystem
  rate-limit     Cisco sshd rate-limit of service requests
  session-limit  Cisco sshd session-limit of service requests
  v2             Cisco sshd force protocol version 2 only
  vrf            Cisco sshd VRF name
  <cr>
RP/0/0/CPU0:XR-2(config)#ssh server v2
RP/0/0/CPU0:XR-2(config)#commit
```

```
XE-1#ssh -l admin 10.1.2.2



IMPORTANT:  READ CAREFULLY
Welcome to the Demo Version of Cisco IOS XRv (the "Software").
The Software is subject to and governed by the terms and conditions
of the End User License Agreement and the Supplemental End User
License Agreement accompanying the product, made available at the
time of your order, or posted on the Cisco website at
www.cisco.com/go/terms (collectively, the "Agreement").
As set forth more fully in the Agreement, use of the Software is
strictly limited to internal use in a non-production environment
solely for demonstration and evaluation purposes.  Downloading,
installing, or using the Software constitutes acceptance of the
Agreement, and you are binding yourself and the business entity
that you represent to the Agreement.  If you do not agree to all
of the terms of the Agreement, then Cisco is unwilling to license
the Software to you and (a) you may not download, install or use the
Software, and (b) you may return the Software as more fully set forth
in the Agreement.



Please login with any configured user/password, or cisco/cisco




Password:



RP/0/0/CPU0:XR-2#
```

You can then use **show ssh session details** to view information about the SSH session being established on the IOS XR router. In addition, you can view TCP-related information for SSH sessions by using the **show tcp brief** command. For SSH services to run properly, you need to ensure that the IOS XR device is listening on port 22. Example 9-12 shows SSH session information using both of these commands.

Example 9-12 *Verifying SSH Sessions*

```
RP/0/0/CPU0:XR-2#show ssh session details
Sun Oct 13 04:44:12.622 UTC
SSH version : Cisco-2.0

id   key-exchange  pubkey  incipher  outcipher  inmac   outmac
---------------------------------------------------------------------
Incoming Session
0   diffie-hellman ssh-rsa aes128-ctr aes128-ctr hmac-sha1  hmac-sha1
```

```
Outgoing connection


RP/0/0/CPU0:XR-2#show tcp brief
Sun Oct 13 04:44:32.821 UTC
   PCB      VRF-ID     Recv-Q Send-Q Local Address   Foreign Address        State
0x1214c250 0x60000000      0      0  :::22          :::0                   LISTEN
0x1214853c 0x00000000      0      0  :::22          :::0                   LISTEN
0x12150f80 0x60000000      0      0  10.1.2.2:22    10.1.2.1:27233         ESTAB
0x1214be00 0x60000000      0      0  0.0.0.0:22     0.0.0.0:0              LISTEN
0x1213beec 0x00000000      0      0  0.0.0.0:22     0.0.0.0:0              LISTEN
```

You can debug an SSH connection on a server enabled on IOS XR by using the command **debug ssh server**. This **debug** command displays all the exchanges that occur between the server and the client and when the SSH client is authenticated on the SSH server. Example 9-13 illustrates the debugging of an incoming SSH connection on an SSH server running on IOS XR software and highlights some of the important information being exchanged.

Example 9-13 *Debugging SSH Server Connections on IOS XR*

```
RP/0/0/CPU0:XR-2#debug ssh server
Sun Oct 13 04:52:09.479 UTC
RP/0/0/CPU0:Oct 13 04:52:15.979 : SSHD_[1133]: sshd_conn_handler:623 type: 1,
port: 22
RP/0/0/CPU0:Oct 13 04:52:15.979 : SSHD_[1133]: ratelimit_msecs:1000.000000,
ratelimit_count:1, low_rate:0
RP/0/0/CPU0:Oct 13 04:52:15.979 : SSHD_[1133]: elapsed:556171.900000,
msecs:1000.000000, count:1
RP/0/0/CPU0:Oct 13 04:52:16.019 : SSHD_[1133]: Spawned new child process 921787
RP/0/0/CPU0:Oct 13 04:52:16.039 : SSHD_[65723]: Inside init_ttylist FUNC
...
RP/0/0/CPU0:Oct 13 04:52:16.189 : SSHD_[65723]: After setting socket options,
sndbuf65536, rcvbuf - 65536
RP/0/0/CPU0:Oct 13 04:52:16.189 : SSHD_[65723]: Connection from 10.1.2.1 port
25781
RP/0/0/CPU0:Oct 13 04:52:16.189 : SSHD_[65723]: Inside sshd_session_sem_create
FUNC
RP/0/0/CPU0:Oct 13 04:52:16.189 : SSHD_[65723]: sshd_session_sem_create:
Created/Opened the Semaphore %pid:921787 SEM Value:1:
RP/0/0/CPU0:Oct 13 04:52:16.189 : SSHD_[65723]: main: Inside the Critical
Section: session_pid=921787, channel_id=1
RP/0/0/CPU0:Oct 13 04:52:16.189 : SSHD_[65723]: (addrem_ssh_info_tuple) ADD
tty:XXXXX(XXXXX), user:()
RP/0/0/CPU0:Oct 13 04:52:16.189 : SSHD_[65723]: main: Exiting the Critical
Section: session_pid=921787, channel_id=1
RP/0/0/CPU0:Oct 13 04:52:16.189 : SSHD_[65723]: Session id 0
RP/0/0/CPU0:Oct 13 04:52:16.189 : SSHD_[65723]: Exchanging versions
RP/0/0/CPU0:Oct 13 04:52:16.199 : SSHD_[65723]: ssh_version_exchange : client_str
= SSH-1.99-Cisco-1.25  (len = 19)
RP/0/0/CPU0:Oct 13 04:52:16.199 : SSHD_[65723]: Remote protocol version 1.99,
remote software version Cisco-1.25
RP/0/0/CPU0:Oct 13 04:52:16.199 : SSHD_[65723]: In Key exchange
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Pad_len = 4, Packlen = 308
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Calling Receive kexinit 10
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Peer Proposal : diffie-hellman-
group-exchange-sha1,diffie-hellman-group14-sha1
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Peer Proposal : ssh-rsa
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Peer Proposal : aes128-
ctr,aes192-ctr,aes256-ctr
```

```
...
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Peer Proposal :
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Negotiated Alg : diffie-hellman-
group14-sha1
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Publikey Alg = ssh-rsa
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Incoming cipher = aes128-ctr
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Outgoing cipher = aes128-ctr
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Incoming mac = hmac-sha1
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Outgoing mac = hmac-sha1
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Keylen Reqd  = 20
RP/0/0/CPU0:Oct 13 04:52:16.209 : SSHD_[65723]: Waiting for KEXDH_INIT
RP/0/0/CPU0:Oct 13 04:52:16.299 : SSHD_[65723]: Pad_len = 6, Packlen = 268
RP/0/0/CPU0:Oct 13 04:52:16.299 : SSHD_[65723]: Received KEXDH_INIT
RP/0/0/CPU0:Oct 13 04:52:16.299 : SSHD_[65723]: Calling DH algorithm setting with
group14
RP/0/0/CPU0:Oct 13 04:52:16.299 : SSHD_[65723]:  Getting the parameter inside
(Func: set_dh_param_groups)
RP/0/0/CPU0:Oct 13 04:52:16.299 : SSHD_[65723]: After geting the parameter we are
calling the first phase of DH
RP/0/0/CPU0:Oct 13 04:52:16.489 : SSHD_[65723]: sshd_key_exchange: Selected
key_type is RSA
RP/0/0/CPU0:Oct 13 04:52:16.489 : SSHD_[65723]: Extracting RSA pubkey from crypto
engine
RP/0/0/CPU0:Oct 13 04:52:16.489 : SSHD_[65723]: Retreiving 2048 bit RSA host key-
pair
RP/0/0/CPU0:Oct 13 04:52:16.489 : SSHD_[65723]: bloblen = 279
RP/0/0/CPU0:Oct 13 04:52:16.489 : SSHD_[65723]: exponent = 3, modulus = 257
RP/0/0/CPU0:Oct 13 04:52:16.489 : SSHD_[65723]: Calculating kex hash with
client_str = SSH-1.99-Cisco-1.25  (len = 19)
RP/0/0/CPU0:Oct 13 04:52:16.489 : SSHD_[65723]: server_str = SSH-2.0-Cisco-2.0
(len = 17)
RP/0/0/CPU0:Oct 13 04:52:16.509 : SSHD_[65723]: Sending KEXDH_REPLY
RP/0/0/CPU0:Oct 13 04:52:16.509 : SSHD_[65723]: Sending NEWKEYS
...
RP/0/0/CPU0:Oct 13 04:52:16.629 : SSHD_[65723]: User:admin,service:ssh-
connection,Method:none
RP/0/0/CPU0:Oct 13 04:52:16.639 : SSHD_[65723]: (sshd_authenticate) Searching RSA
key for user:admin
RP/0/0/CPU0:Oct 13 04:52:16.639 : SSHD_[65723]: (sshd_authenticate) user:admin,
rsa public key not found
RP/0/0/CPU0:Oct 13 04:52:16.639 : SSHD_[65723]: (sshd_authenticate) setting alarm
to 30 secs
RP/0/0/CPU0:Oct 13 04:52:16.639 : SSHD_[65723]: Waiting for Userauth req
RP/0/0/CPU0:Oct 13 04:52:19.409 : SSHD_[65723]: Pad_len = 9, Packlen = 60
RP/0/0/CPU0:Oct 13 04:52:19.409 : SSHD_[65723]: (sshd_authenticate) removing
alarm
RP/0/0/CPU0:Oct 13 04:52:19.409 : SSHD_[65723]: User:admin,service:ssh-
connection,Method:password
RP/0/0/CPU0:Oct 13 04:52:19.409 : SSHD_[65723]: Has password expired:NO
RP/0/0/CPU0:Oct 13 04:52:19.409 : SSHD_[65723]: sshd_authenticate_internal:
username:admin, method:PASSWORD
RP/0/0/CPU0:Oct 13 04:52:19.409 : SSHD_[65723]: sshd_authenticate_internal:
sshd.ch[id].ttyname:XXXXX, INIT_VAL:XXXXX
RP/0/0/CPU0:Oct 13 04:52:19.409 : SSHD_[65723]: sshd_authenticate_internal:
```

```
Calling allocpty
RP/0/0/CPU0:Oct 13 04:52:19.409 : SSHD_[65723]: Connecting to VTY Server, dest
port:16, src port:64b5
< SNIP >
RP/0/0/CPU0:Oct 13 04:52:19.439 : SSHD_[65723]: sshd_after_authenticate: Shared
memory task table copied at(0x402880ac), authen_method:2, groups:root-system,
cisco-support
RP/0/0/CPU0:Oct 13 04:52:19.439 : SSHD_[65723]: sshd_aaa_account:no aaa
accounting cfg
RP/0/0/CPU0:Oct 13 04:52:19.439 : SSHD_[65723]: Done with AAA APIs
RP/0/0/CPU0:Oct 13 04:52:19.439 : SSHD_[65723]: In login_success_banner_msg
RP/0/0/CPU0:Oct 13 04:52:19.449 : SSHD_[65723]: (addrem_ssh_info_tuple) UPDATE
session_id:0, channel_id: 1, tty:vty0(vty0), user:admin(admin), connection_type:
0
RP/0/0/CPU0:Oct 13 04:52:19.449 : SSHD_[65723]: In Interactive shell
! Output omitted for brevity
```

Configuring SSH on NX-OS

On Nexus OS (NX-OS) software, an SSH2 server is enabled by default. In addition, the NX-OS software generates an RSA key with a modulus of 1024 bits when it boots up for the first time, which makes it possible to enable SSH-based login on any NX-OS device. You can generate an RSA key with a higher modulus or other algorithms, such as DSA or Elliptic Curve DSA (ECDSA). NX-OS supports key generation with a maximum modulus of 2048 bits. To validate whether SSH has been enabled, you use the command **show feature** and filter the command output for the **sshServer** process. You use the **show ssh server** command to validate whether the SSH server has been enabled. Once you have validated that the SSH server is enabled, the next step is to verify that the RSA key is generated and, if required, generate a key with a larger size. The **show ssh key [rsa | dsa | ecdsa]** command displays the keys generated using different methods, along with a SHA-256 fingerprint. The **show ssh key md5** command, on the other hand, displays the generated key with its MD5 fingerprint. Example 9-14 shows how to verify the SSH feature and its default generated key.

Example 9-14 *Verifying the SSH Feature on NX-OS*

```
NX-3# show feature | include ssh
sshServer             1          enabled

NX-3# show ssh server
ssh version 2 is enabled

NX-3# show ssh key rsa
**************************************
rsa Keys generated:Mon Sep  3 22:04:53 2018

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDGXZo9uA6PNW7fMU9WlsL2MWHcQjXOWJghLQRbCNOQ
o8cr9QCEYdl1QxY01AOVRJuKckoNghOPWIbrc79rIcvFZ1TUbUfWUYR9KwaDfy+WuyLiTZz+Uzt4t6zI
AxOcewGCr6fHgyIU95xHPBiez0JhmbdSmOt5plMwAh4GLuDErQ==

bitcount:1024
fingerprint:
SHA256:5vWe6hoXV+PImUOSE2K5q+pqO7gI5pKM0UT0N71z1Ko

NX-3# show ssh key md5
**************************************
rsa Keys generated:Mon Sep  3 22:04:53 2018

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDGXZo9uA6PNW7fMU9WlsL2MWHcQjXOWJghLQRbCNOQ
o8cr9QCEYdl1QxY01AOVRJuKckoNghOPWIbrc79rIcvFZ1TUbUfWUYR9KwaDfy+WuyLiTZz+Uzt4t6zI
AxOcewGCr6fHgyIU95xHPBiez0JhmbdSmOt5plMwAh4GLuDErQ==

bitcount:1024
fingerprint:
MD5:8d:a9:d5:e3:fa:2a:71:2d:f3:7d:53:02:e3:26:e0:37
```

```
*************************************
could not retrieve dsa key information
*************************************
could not retrieve ecdsa key information
*************************************
```

Because a 1024-bit RSA key is not the most secure key for use with SSH, you can generate a 2048-bit SSH key by using the configuration command **ssh key rsa 2048** [**force**]. Note that the new key can be generated only once the SSH feature has been disabled. After generating the new key, you can turn the SSH feature back on by using the **feature ssh** configuration command. Example 9-15 demonstrates how to generate a key with a larger bit size and validate it with the **show ssh key rsa** command.

Example 9-15 *Generating and Verifying a NewRSA key*

```
NX-3(config)# ssh key rsa 2048
rsa keys already present, use force option to overwrite them

NX-3(config)# ssh key rsa 2048 force
deleting old rsa key.....
ssh server is enabled, cannot delete/generate the keys
NX-3(config)# no feature ssh
XML interface to system may become unavailable since ssh is disabled
NX-3(config)#
NX-3(config)# ssh key rsa 2048 force
deleting old rsa key.....
generating rsa key(2048 bits).....
....
generated rsa key
NX-3(config)# feature ssh
NX-3(config)# end
```

```
NX-3# show ssh key rsa
*************************************
rsa Keys generated:Sun Oct 13 07:41:11 2019

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC2Wk5k3+9vEkzE7qC0WAY4nSSndoonS0amc9PiKcMd
uH2/uxFdmFt/TNOyZimzvXzXJSr8nED4aSZtD/HCnitFy7EOLAQo4EsYB4R941BRik9HwOAXlv/Iuokz
irRfKfnVzpTKKQQ2mAoUGJgppl/yHTHQvPAuG+GY8fEGdIUKm7GLooIkLI6skd04va99/U3EzOTShTOG
twfkXws/pc6CrliOhuJZI0YlpkTLvEScE5CFZLmO2bCi8iC/L20WFUokII17TwQ5xu0rGvpcQmW+G70y
VT7YTyt+ops2A0blo/EI6UgZ2KSVDkfM0IRdyt0zXPToMyhbuE9GvxUZoBfL

bitcount:2048
fingerprint:
SHA256:Q+BeW09KdikEKblzakmjyqva6JveGupfByMbD4EF+yw
```

Once a client is connected to an NX-OS device, you can verify the SSH session on port 22 by using the command **show sockets connection tcp** [**detail**]. Note that when the SSH feature is enabled, the device is already in a listening state on port 22 for both IPv4 and IPv6 addresses. Example 9-16 displays the output of the **show sockets connection tcp** command, highlighting the SSH connection from a client machine.

Example 9-16 ***show sockets connection tcp*** *Command Output*

```
NX-3# show sockets connection tcp

Total number of netstack tcp sockets: 4
Active connections (including servers)
        Protocol State/       Recv-Q/   Local Address(port)/
                 Context      Send-Q    Remote Address(port)
[host]: tcp(4/6) LISTEN       0         *(22)
                 Wildcard     0         *(*)

[host]: tcp      LISTEN       0         *(161)
```

```
                Wildcard     0         *(*)


[host]: tcp(4/6) LISTEN       0         *(161)

                Wildcard     0         *(*)


[host]: tcp      ESTABLISHED  0         10.1.3.3(22)

                default      0         10.1.3.1(16343)
```

An NX-OS device not only acts as an SSH server but can also be used as a client. The SSH client software is enabled by default. On NX-OS, you can also establish an SSH connection from the Bash shell. The Bash shell feature provides access to the Linux shell on the system, from which users and network administrators can execute Linux commands. Example 9-17 illustrates how to enable the Bash shell feature and use the Bash shell to establish a new SSH connection or verify open TCP connections used by the **sshd** process.

Example 9-17 *Establishing SSH from the Bash Shell*

```
NX-3(config)# feature bash

NX-3# run bash

bash-4.3$

bash-4.3$ lsof -nP -i TCP | grep ssh

sshd        816    11000    3u  IPv4  93874      0t0  TCP 127.0.0.1:17682 (LISTEN)

sshd      12601     root    3u  IPv6  33279      0t0  TCP *:22 (LISTEN)

sshd      12601     root    4u  IPv4  33281      0t0  TCP *:22 (LISTEN)


bash-4.3$ ssh admin@10.1.3.1

Outbound-ReKey for 10.1.3.1:22

Inbound-ReKey for 10.1.3.1:22

Password:


XE-1#

XE-1#exit

Connection to 10.1.3.1 closed by remote host.

Connection to 10.1.3.1 closed.
```

Note

In Example 9-17, the use of the Bash shell is for demonstration purposes only. It is not recommended to establish SSH sessions from the Bash shell; the Bash shell should be used only for troubleshooting purposes.

Secure File Transfer

Traditionally, file transfers over the network were performed using the legacy insecure File Transfer Protocol (FTP). Developed in 1971 as part of the ARPANET protocols, FTP is one of the original programs for accessing information over the Internet. An extension to FTP, FTP Secure (FTPS), has been developed, and the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) cryptographic protocols have also been added. Some of the common challenges with FTPS are that not all servers support SSL/TLS encryption, and there is not a standard way of fetching and changing file and directory attributes. Some of these challenges are easily overcome by SSH FTP.

SSH FTP (SFPT) is an extension to the already secure SSH protocol that provides easy file transfer capability. The SFTP protocol is designed on SSH standards that strictly define all aspects of operations. When transferring files over SFTP, the connection is already secured by the SSH protocol. SFTP not only deals with file transfer but also includes operations for permission, attribute manipulation, file locking, and other processes. The following are some of the advantages of using SFTP:

• **Data exchange**: The SFTP protocol formats a command and data into a special packet format and sends them through a single secure connection. This means that the server administrator does not need to open additional ports to allow file transfer, which is otherwise required for protocols such as FTP or FTPS.

• **Security**: SFTP includes all the secure functionality of SSH.

• **Operational simplicity**: Because SFTP works over SSH, it is easier to manage firewalls as there is no need to open extra ports for file transfer.

• **Integrity**: Data security and integrity are crucial for any business environment. Performing file transfer over SFTP provides data integrity thanks to SSH.

Many graphical tools, such as FileZilla, are available for configuring SFTP, but the focus of this chapter is on using the command line to configure the SFTP server and client. The first step in setting up SFTP is to ensure that SSH between the server and the client is functional. When you install OpenSSH software, the SFTP application is installed by default. To open an SFTP session, you use the following command:

```
sftp username@remote_server
```

Then, when you log in to the server with the password or SSH key, you are presented with the SFTP prompt. Example 9-18 illustrates the process of logging in to an SFTP server.

Example 9-18 *Performing SFTP Login on a Server*

```
Client1:~ centos$ sftp root@192.168.1.100
```

```
Enter passphrase for key '/Users/centos/.ssh/id_rsa':

Connected to root@167.71.231.196.

sftp>
```

Note

If the server administrator has configured a non-default port number for SSH (TCP/port 22), users can connect to SFTP on a custom port number by using the following command:

```
sftp -oPort=custom_port username@remote_server
```

The SFTP shell provides access to a lot of commands that can be used for performing various operations, such as local and remote directory listing, fetching or uploading a file to the SFTP server, or changing file permissions. You can view a list of command-line options available with SFTP by using the **?** or **help** command under the SFTP shell, as shown in Example 9-19.

Example 9-19 *SFTP Help*

```
sftp> help

Available commands:

bye                                Quit sftp

cd path                            Change remote directory to 'path'

chgrp grp path                     Change group of file 'path' to 'grp'

chmod mode path                    Change permissions of file 'path' to 'mode'

chown own path                     Change owner of file 'path' to 'own'

df [-hi] [path]                    Display statistics for current directory or

                                   filesystem containing 'path'

exit                               Quit sftp

get [-afPpRr] remote [local]       Download file

reget [-fPpRr] remote [local]      Resume download file

reput [-fPpRr] [local] remote      Resume upload file

help                               Display this help text

lcd path                           Change local directory to 'path'

lls [ls-options [path]]            Display local directory listing

lmkdir path                        Create local directory

ln [-s] oldpath newpath            Link remote file (-s for symlink)

lpwd                               Print local working directory

ls [-1afhlnrSt] [path]             Display remote directory listing

lumask umask                       Set local umask to 'umask'

mkdir path                         Create remote directory

progress                           Toggle display of progress meter

put [-afPpRr] local [remote]       Upload file

pwd                                Display remote working directory

quit                               Quit sftp

rename oldpath newpath             Rename remote file

rm path                            Delete remote file

rmdir path                         Remove remote directory

symlink oldpath newpath            Symlink remote file

version                            Show SFTP version

!command                           Execute 'command' in local shell

!                                  Escape to local shell
```

The SFTP shell provides access to local as well as remote directories. Linux shell commands such as **pwd**, **ls**, and **cd** are used to perform file operations on remote servers, and the commands **lpwd**, **lls**, and **lcd** are used to perform file operations on the local file system. The command **pwd** is used to display the remote working directory, and **lpwd** is used to display the local working directory. Example 9-20 demonstrates the use of these commands for local and remote directory operations.

Example 9-20 *Local and Remote Directory Operations via SFTP*

```
sftp> pwd

Remote working directory: /root

sftp> ls

anaconda-ks.cfg   original-ks.cfg

sftp> lpwd
```

```
Local working directory: /Users/centos

sftp> lls

Applications            Downloads               Music                   Public

Box Sync                Dropbox                 MyJabberFiles           octave

Desktop                 Library                 Pictures                pgadmin.log

Documents               Movies                  Postman                 vmmaestro

sftp> cd /etc

sftp> pwd

Remote working directory: /etc

sftp> lcd Desktop

sftp> lpwd

Local working directory: /Users/centos/Desktop

sftp>
```

Once the local and remote paths have been set, the next step is to either fetch a remote file and save it on a local directory or upload a file from a local directory to a remote directory. This can be done using the **get** and **put** commands, as shown in Example 9-21. In this example, the command **get anaconda-ks.cfg test.cfg** fetches the file anaconda-ks.cfg from the remote SFTP server and saves it with the name test.cfg in the local directory. Similarly, the **put test.cfg** command is used to upload the test.cfg file from the local directory to the currently selected remote directory and save it with the same name.

Example 9-21 *Fetching and Uploading Files via SFTP*

```
sftp> ls

anaconda-ks.cfg   original-ks.cfg

sftp> get anaconda-ks.cfg test.cfg

Fetching /root/anaconda-ks.cfg to test.cfg

/root/anaconda-ks.cfg                                 100% 6921     9.4KB/s   00:00


sftp> lls -al test.cfg

-rw-------  1 vinijain  staff  6921 Oct 18 02:35 test.cfg


sftp> put test.cfg

Uploading test.cfg to /root/test.cfg

test.cfg                                              100% 6921     1.9KB/s   00:03

sftp> ls

anaconda-ks.cfg   original-ks.cfg   test.cfg

sftp>
```

The SFTP shell also provides access to commands that allow for changing file permissions. For instance, you can change remote file permissions by using the Linux **chmod** command. Similarly, you can change the user group access of a file by using the Linux **chgrp** command from the SFTP shell. The **chgrp** Linux command takes a parameter as the group ID of the group that is assigned to the file permissions. To fetch the group ID, you can first get the /etc/group file from the server onto the local directory and then use **!** along with the **less** command to read the content of the file named group. Once the group ID has been identified, the same value can be used with the **chgrp** command, as shown in Example 9-22. Finally, you can use the **bye** or **exit** command to exit the SFTP shell.

Example 9-22 *Changing File Permissions in SFTP*

```
sftp> ls -l

-rw-------    1 root     root         6921 Aug  8 12:48 anaconda-ks.cfg

-rw-------    1 root     root         6577 Aug  8 12:48 original-ks.cfg

-rw-------    1 root     root         6921 Oct 18 10:27 test.cfg

sftp> chmod 777 test.cfg

Changing mode on /root/test.cfg

sftp> ls -l

-rw-------    1 root     root         6921 Aug  8 12:48 anaconda-ks.cfg

-rw-------    1 root     root         6577 Aug  8 12:48 original-ks.cfg

-rwxrwxrwx    1 root     root         6921 Oct 18 10:27 test.cfg
```

```
sftp> get /etc/group

Fetching /etc/group to group

/etc/group                                          100%  602     1.0KB/s   00:00

sftp> !less group

root:x:0:
```

```
bash-4.3$

bash-4.3$ dcos_sftp

usage: dcos_sftp [-1246aCfpqrv] [-B buffer_size] [-b batchfile] [-c cipher]

          [-D sftp_server_path] [-F ssh_config] [-i identity_file] [-l limit]

          [-o ssh_option] [-P port] [-R num_requests] [-S program]

          [-s subsystem | sftp_server] host

       dcos_sftp [user@]host[:file ...]

       dcos_sftp [user@]host[:dir[/]]

       dcos_sftp -b batchfile [user@]host

bash-4.3$ which dcos_sftp

/isan/bin/dcos_sftp
```

Secure Copy Protocol

Secure Copy Protocol (SCP) is a protocol and a tool that works on top of SSH and enables you to copy files between local and remote devices or between two remote devices. SCP runs on port 22 and behaves somewhat like FTP—but with security and authentication. SCP also benefits from SSH as it allows the inclusion of both permissions and time stamps for files. Much like SFTP, SCP can be used to download files or even directories containing files. The **scp** command-line tool is native to most operating systems, including Linux, macOS, and Windows. Example 9-27 illustrates how to use the **scp** tool to copy files to and from a server and between two remote servers.

Example 9-27 *SCP on CentOS*

```
[root@centos2 centos]# scp root@10.1.100.100:test.txt new-test.txt

Enter passphrase for key '/root/.ssh/id_rsa':

test.txt                                      100%    0     0.0KB/s   00:00
```

```
[root@centos2 centos]# scp new-test.txt root@10.1.100.100:frm-centos2.txt

Enter passphrase for key '/root/.ssh/id_rsa':

new-test.txt                                  100%    0     0.0KB/s   00:00
```

```
[root@centos1 ~]# scp root@10.1.100.100:test.txt root@10.1.101.101:new-test2.txt

The authenticity of host '10.1.100.100 (10.1.100.100)' can't be established.

ECDSA key fingerprint is SHA256:6RbYWRplqpY/1Rxg8BgYiEKllxiN5cC5JP36ChDcYo8.

ECDSA key fingerprint is MD5:8f:26:54:06:12:b0:c0:8d:1a:ad:ff:8c:80:c9:9a:d8.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added '10.1.100.100' (ECDSA) to the list of known hosts.

root@10.1.100.100's password:

Enter passphrase for key '/root/.ssh/id_rsa':

test.txt                                      100%    0     0.0KB/s   00:00

Connection to 10.1.100.100 closed.
```

Note

The **scp** command is also available on Cisco network operating systems. You can either use it with the **copy** command (as with SFTP) on both Cisco IOS XE or NX-OS or use the **scp** command on IOS XR to perform SCP operations. For using SCP, it is important that the remote devices are SSH enabled.

Although SCP and SFTP are both based on SSH, there are a few similarities and dissimilarities. Table 9-1 compares SCP and SFTP.

Table 9-1 *SCP and SFTP Comparison*

Feature	SFTP	SCP
Speed	SFTP is comparatively slower than SCP. SFTP much acknowledge every single packet that is exchanged between client and server.	SCP is much faster than SFTP. SCP doesn't require acknowledgements for all packets exchanged.
Functionality	SFTP can easily perform other directory operations, such as listing a directory or creating or deleting a directory, along with performing file transfers.	SCP is purely built for file transfer. It cannot perform operations such as directory listings or other directory operations.
Security	SFTP provides the same level of security as SSH.	SCP provides the same level of security as SSH.
Resuming file transfers	SFTP makes it possible to resume interrupted file transfers by using the **-a** option with the **sftp** command.	SCP does not allow you to resume interrupted file transfers.

You need to choose either the SCP or SFTP protocol based on the use case. For example, if faster transfer of data is important, then SCP is a better option, but if you have an unstable connection to the remote server, SFTP might be a better option.

Summary

This chapter covers SSH in detail, including the following topics:

• How SSH works and some of the benefits SSH provides over legacy protocols such as Telnet and rcp

• The differences between SSH1 and SSH2

• The three other protocols defined in the SSH2 RFCs that work together to provide the functionality of SSH2: SSH Transport Layer Protocol, SSH Connection Protocol, and SSH Authentication Protocol

• How to set up SSH on both Linux servers and Cisco devices

• SSH capabilities to securely transfer files to a remote server via protocols such as SFTP and SCP running over SSH

References

RFC 4251, "The Secure Shell (SSH) Protocol Architecture," https://tools.ietf.org/html/rfc4251.

RFC 4252, "The Secure Shell (SSH) Authentication Protocol," https://tools.ietf.org/html/rfc4252.

RFC 4253, "The Secure Shell (SSH) Transport Layer Protocol," https://tools.ietf.org/html/rfc4253.

RFC 4254, "The Secure Shell (SSH) Connection Protocol," https://tools.ietf.org/html/rfc4254.

Part V: Encoding

Chapter 10. XML and XSD

This chapter covers the first encoding format listed in the network programmability stack: Extensible Markup Language (XML). XML is used to encode the messages of network programmability protocols. Some protocols, such as NETCONF, support only XML. XML was the first data representation language to be developed in the XML/JSON/YAML family. As you will see in this chapter, not only is XML the oldest of the three encoding schemes, but it is also the most powerful. In this chapter, you will also learn the details of XSD (XML Schema Definition), which is used to improve the capabilities of XML for complex applications.

XML Overview, History, and Usage

XML is a meta-markup language created to deal with information structures. "Meta" refers to the fact that XML has the ability to use metadata for better information handling, and "markup" means that it uses various *tags* to define the structure of the data. XML is designed to be both human readable and machine readable; it is easily readable and modifiable by humans, and when properly tagged, it is easily readable by machines.

In the world of network automation and orchestration, the data structure is one of the key elements. In fact, for all distributed applications involving any data exchange between multiple components, the data structure is crucial. The primary objective of XML is to create a proper data structure for data storage and representation. In 1998, during a time when the Internet was being developed extensively, XML was developed to complement HTML, which is focused on data visualization. XML enables a user create any data structure, in the sense that it doesn't have any predefined tag names or values. In addition, there are limitations in terms of the data types supported by XML documents. This flexibility is very useful during the creation of applications; however, it is important to have a mechanism to check the content of the values to prevent situations such as data within an XML document being out of the range supported by the application. Therefore, further development of XML led to the introduction of XML DTD (Document Type Definition) and XML schemas, which add strict constraints to the content of XML documents to make it more formal.

The latest XML standard, released in 2008, is supported by the World Wide Web Consortium (W3C), which is in charge of updating the standard (https://www.w3.org/standards/xml/core). The IETF outlined the general guidelines of XML usage for development of IETF protocols in RFC 3470 in 2003.

Despite being a relatively old language, XML is still widely used for network automation and application communication. In network automation, the most popular application of XML is the management of network elements over NETCONF (as discussed in Chapter 14 "NETCONF and RESTCONF"). The payload of a NETCONF message is an XML document, which contains all the information needed to configure a network element or get its state.

For application-to-application communication, an XML-based protocol called SOAP (Simple Object Access Protocol) was also developed and supported by the W3C. Whereas NETCONF operates over SSH, SOAP relies on HTTP. (SOAP specifications were maintained by the XML Protocol Working Group of the W3C until the group was closed in 2009.) The REST framework, which is more commonly based on JSON data encoding, is now more commonly used than SOAP for communication between applications (see Chapter 11, "JSON and JSD").

XML Syntax and Components

XML is very flexible in terms of the data structure as all the tags you use you create yourself upon creating the structure of the data. With a NETCONF application, XML structure follows the internal structure of the database where the network operating system stores its configuration; in some cases it could be similar to the command-line interface (CLI) hierarchy, whereas in others it may vary. Despite the great flexibility, there are certain formatting rules that you need to follow to make XML syntax correct.

XML Document Building Blocks

To familiarize you with XML, Example 10-1 shows an XML document.

Example 10-1 *Basic XML Document*

```
<?xml version="1.0" encoding="UTF-8"?>

<vpn>

  <customer>NPF_cust1</customer>

  <contact>info@npf.cust1</contact>

</vpn>
```

In XML syntax, a *tag* is framed by < and > characters. There are two type of tags—the *start tag* (for example, <vpn>) and the *end tag* (for example, </vpn>)—and they are always paired. A storage unit framed by these tags is called an *element*. The topmost element is called a *root element*, and there can be only one root element per XML document.

Everything between the start and end tags is the *value* associated with the key represented by the pair of tags. Typically everything in XML should have a start tag and an end tag, but there are two exceptions to this rule:

• The document element, which is the first element in the document, starts with <?xml and ending with ?>. This element declares that a file is an XML document, and the XML processer should be used. Per W3C recommendation, the version must always be set to 1.0, and encoding must be either UTF-8 or UTF-16, with UTF-8 being the default choice (see https://www.w3resource.com/xml/declarations.php).

• The empty element tag is a tag for an element that contains no value, though it might contain an attribute. (You will learn about attributes later in this chapter.) However, in the XML encoding of some Cisco IOS XR YANG modules, you might find the empty element tag without any attribute.

In Example 10-1, you can see that the XML document is based on a parent/child relationship. The tag <vpn> is a parent for both the <customer> and <contact> entries, and <contact> and <customer> are children of <vpn>. Typically, child elements are visually marked by some spaces at the beginning of the line, which are ignored by XML processor. Example 10-2 shows the representations of data that have the same meaning for an XML processor.

Example 10-2 *Leading Spaces in XML Document*

```
! Option 1

<?xml version="1.0" encoding="UTF-8"?>

<vpn>

  <customer>NPF_cust1</customer>
```

```
  <contact>info@npf.cust1</contact>
</vpn>



! Option 2
<?xml version="1.0" encoding="UTF-8"?>
<vpn><customer>NPF_cust1</customer><contact>info@npf.cust1</contact></vpn>
```

There is no limit on the number of the parent/child relationships in an XML document, as shown in Example 10-3, which extends the XML document from Example 10-1.

Example 10-3 *XML Document with Multiple Nesting Relationships*

```
<?xml version="1.0" encoding="UTF-8"?>
<vpn>
  <customer>NPF_cust1</customer>
  <contact>info@npf.cust1</contact>
  <sites>
    <site id='1'>
      <provider_side>
        <router>DUS-1</router>
        <port>Gig1/1/1</port>
        <ipv4>10.0.0.1/30</ipv4>
        <ipv6>fc00::10:0:0:1/126</ipv6>
      </provider_side>
      <customer_side>
        <router>CUST-DUS-1</router>
        <port>xe-0/0/1</port>
        <ipv4>10.0.0.2/30</ipv4>
        <ipv6>fc00::10:0:0:2/126</ipv6>
      </customer_side>
      <enabled/>
    </site>
    <site id='2'>
      <provider_side>
        <router>FRA-1</router>
        <port>1/1/2</port>
        <ipv4>10.0.0.5/30</ipv4>
        <ipv6>fc00::10:0:0:5/126</ipv6>
      </provider_side>
      <customer_side>
        <router>CUST-FRA-1</router>
        <port>Ethernet1</port>
        <ipv4>10.0.0.6/30</ipv4>
        <ipv6>fc00::10:0:0:6/126</ipv6>
      </customer_side>
      <enabled/>
    </site>
  </sites>
</vpn>
```

As you can see in Example 10-3, multiple entities might have the same name within a single parent, each for an array or a list data structure. In total, all the children form a tree structure starting from the root element. In this example, the tag <enabled/> is an empty element tag.

You can create your own tags, but it is important to following several naming conventions:

- A tag name can contain any alphanumeric value as well as symbols (for example, -, _, :, and .).
- A tag cannot start with a number or a symbol (such as - or .).
- A tag cannot start with any variation of XML, such as xml, XML, or XmL.
- Tags are case sensitive; therefore, for example, <vpn> and <Vpn> are two completely different tags.

```
        <ipv6>fc00::10:0:0:1/126</ipv6>
      </provider_side>
      <customer_side>
        <router>CUST-DUS-1</router>
        <port>xe-0/0/1</port>
        <ipv4>10.0.0.2/30</ipv4>
        <ipv6>fc00::10:0:0:2/126</ipv6>
      </customer_side>
      <enabled/>
    </site>
  </sites>
  <sites>
    <site>
      <address>
        <country>Germany</country>
        <town>Berlin</town>
        <street>Unter den Linden</street>
        <house>78</house>
      </address>
    </site>
  </sites>
</vpn>
```

In Example 10-5, note that there are two <sites> tags that contain completely different data: The first one is focused on the technical information required to create a customer BGP IP MPLS VPN, whereas the second one contains the physical address. It is possible that there might be a mistake in the tag name; alternatively, the provided XML file might be used by different programs, where different information is needed. There is a way to use the same tag name in different ways within a single document: by using namespaces. An XML namespace is a specific attribute that defines XML vocabulary used to process a specific part of the XML document. Example 10-6 adds the XML namespace to the XML document from Example 10-5.

Example 10-6 *Using XML Namespaces*

```
<?xml version="1.0" encoding="UTF-8"?>
<vpn xmlns="http://network.programmability/xmldocs/namespace1">
  <customer>NPF_cust1</customer>
  <contact>info@npf.cust1</contact>
  <sites>
    <site id='1'>
    <!-- output is truncated for brevity -->
  </sites>
  <sites xmlns="http://network.programmability/xmldocs/namespace2">
    <site>
    <!-- output is truncated for brevity -->
  </sites>
</vpn>
```

In this example, the xmlns attribute's value is a URI (uniform resource indicator). xmlns is inherited, which means each child entry inherits the XML namespace from its parent. The URI is processed by the XML parser as a string, and it doesn't necessarily represent a real web page you can access on the Internet (though it might). Typically a parser uses the xmlns attribute to convey data to the corresponding software for further processing.

An application can process XML namespaces properly, but for humans, it might be hard to follow all the parent/child relationships and understand which dictionary the particular elements belongs to. To overcome this ambiguity, XML introduced the concept of prefixes, which are used to explicitly map a certain tag to a certain namespace, as illustrated in Example 10-7.

Example 10-7 *Using XML Namespaces with Prefixes*

```
<?xml version="1.0" encoding="UTF-8"?>
<prl:vpn xmlns:prl="http://network.programmability/xmldocs/namespace1"
xmlns:pr2="http://network.programmability/xmldocs/namespace2">
  <prl:customer>NPF_cust1</prl:customer>
  <prl:contact>info@npf.cust1</prl:contact>
  <prl:sites>
    <prl:site id='1'>
```

```
    <!-- output is truncated for brevity -->
  <pr1:/sites>
  <pr2:sites>
    <pr2:site>
    <!-- output is truncated for brevity -->
  </pr2:sites>
</pr1:vpn>
```

A prefix is defined during the declaration of an XML namespace as xmlns:*prefix*—for example, xmlns:pr1 and xmlns:pr2 in Example 10-7. After the namespace has been defined, all the tags must start with an appropriate prefix value the form *prefix*:*tag*—for example, pr1:vpn or pr2:sites. In Example 10-7,you can see that both of the namespaces are declared in the root element and then mapped to the respective tags using prefixes.

The W3C does not mandate the use of XML namespaces. If neither default namespace (for example, xmlns="*URI*") nor a namespace with a prefix (for example, xmlns:*prefix*="*URI*") is defined, then element is implicitly not related to any XML namespace. However, some protocols that rely on XML (for example, NETCONF) require the xmlns attribute to be present starting from the root element.

XML Formatting Rules

There are some more rules that you need to be aware of when you work with XML. First of all, you must always strictly follow the rules for parent/child relationships, which create your data structure. Example 10-8 shows both correct and incorrect XML documents.

Example 10-8 *XML Nesting Format*

```
! Incorrect example
<?xml version="1.0" encoding="UTF-8"?>
<vpn>
  <customer>NPF_cust1</customer>
  <contact>info@npf.cust1</contact>
  <sites>
    <site id='1'>
      <!-- Details -->
    </sites>
  </site>
</vpn>



! Correct example
<?xml version="1.0" encoding="UTF-8"?>
<vpn>
  <customer>NPF_cust1</customer>
  <contact>info@npf.cust1</contact>
  <sites>
    <site id='1'>
      <!-- Details -->
    </site>
  </sites>
</vpn>
```

In the first part of Example 10-8, the end tags </sites> and </site> are incorrectly placed. This type of mistake can easily be found by using a professional XML editor, but with a console text editor (for example, vim on Linux), it is very easy to make such a mistake. If you create your own data structure, you can avoid this mistake by creating proper tag names. Unfortunately, in many established YANG modules (for example, Cisco IOS XR), several adjacent tags are very similar (for example, the tag <interfaces> is the parent to <interface>). Incorrectly sequencing parent and child tags will result in an error in XML document processing.

Another important thing you should know about XML format is that everything between the start tag and end tag is a value. If there are leading or trailing spaces, as in Example 10-9, they are considered values.

Example 10-9 *Space Characters in the Value Fields*

```
! Incorrect example
<?xml version="1.0" encoding="UTF-8"?>
<vpn>
  <customer> NPF_cust1</customer>
  <contact>info@npf.cust1  </contact>
</vpn>
```

In the XML document in Example 10-9, the key customer has the value *NPF_cust1*, starting with a leading space, and the key contact the value

info@npf.cust1 with two trailing spaces. Because these spaces are considered part of the value, when the XML document is processed, any operations involving comparing the values with others may result in undesired results.

If you follow all the rules described so far in this chapter, you will create *well-formed* XML documents. An XML document must be well formed in order to be processed correctly. If an XML document contains syntax violations, it is not well formed. Such a document cannot be properly processed by an XML processor, and any application that. relies on the document may fail.

With XML-based applications, a whole XML document must be parsed, analyzed, and processed before any further activity (such as changes to a network element) can be accomplished. It is therefore recommended that you keep XML documents reasonably small. Otherwise, the performance of the applications could be severely affected.

Making XML Valid

In addition to being well formed, an XML document must comply with another condition: It must be *valid*. *Validity* means that a document is properly formatted and that it contains the proper content. To be valid, a document must be well formed, and it must comply with the guidelines for the document type as expressed by the XML DTD file or the XML schema.

One of the primary goals of ensuring that XML documents are valid is to reduce the number of errors that are caused by improper tags or data types in an XML document. As mentioned earlier, XML documents with the errors cannot be processed.

XML DTD

An XML Document Type Definition (DTD) defines the structure and the legal elements and attributes of an XML document. It contains or points to markup declarations that provide the grammar for a class of documents. A DTD can point to an external entity that contains markup declarations, or it can contain the markup declarations directly in an internal entity, or both. In this context, a markup declaration is an element type declaration, an attribute list declaration, an entity declaration, or a notation declaration. Basically, these declarations define which elements (that is, tags) are allowed in a specific XML document and what values they may take.

Example 10-10 extends the XML document from Example 10-1 to point to an external XML DTD document.

Example 10-10 *Basic XML Document with DTD*

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE vpn SYSTEM "vpn.dtd">

<vpn>

  <customer id="1">NPF_cust1</customer>

  <contact>info@npf.cust1</contact>

</vpn>
```

The DTD in this example starts with *<!DOCTYPE* followed by the path to the file. In Example 10-10, it is expected that the file vpn.dtd is located in the same folder as the original XML document. The DTD document should include all the tags with the data type expected in a validated XML document. Example 10-11 shows what such a DTD file might look like.

Example 10-11 *Sample DTD File*

```
<!DOCTYPE vpn [

<!ELEMENT vpn (customer,contact)>

<!ELEMENT customer (#PCDATA)>

<!ATTLIST customer id ID #REQUIRED>

<!ELEMENT contact (#PCDATA)>

]>
```

The first line of this document defines the root element of the validated document. In Example 10-10, the root element has a <vpn> tag; therefore, !DOCTYPE has the value vpn as well.

The following entries that start with !ELEMENT define the content of the tags in the original XML document:

- The element vpn consists of two other elements: customer and contact.
- The element customer must have the #PCDATA type.
- The element contact must have the #PCDATA type as well.

Note

The term PCDATA derives historically from the term *parsed character data*, which could be any kind of input data.

There is another entry type defined by the !ATTLIST command, which validates the attributes attached to a certain tag.

The DTD in Example 10-11 checks the following in the original XML document:

- It ensures that the root element is vpn.
- It ensures that the element vpn has exactly two children: customer and contact.
- It ensures that the each of the children (customer and contact) has some kind of textual information, including a zero string (for when there is no input).
- It ensures that the tag customer has the attribute id.

As mentioned earlier in this section, the DTD might be located in a separate file, or it might be part of the initial XML document. A joint XML document with DTD looks as shown in Example 10-12.

Example 10-12 *Joint XML and DTD File*

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE vpn [

<!ELEMENT vpn (customer,contact)>

<!ELEMENT customer (#PCDATA)>

<!ELEMENT contact (#PCDATA)>

<!ENTITY fixed_info "Configured automatically">

]>

<vpn>

  <customer>NPF_cust1</customer>

  <contact>info@npf.cust1</contact>

  <hint>&fixed_info</hint>

</vpn>
```

In addition to showing a joint XML and DTD document, Example 10-12 highlights the additional capability of the XML DTD to define some information (which is likely fixed) in the DTD part that can be inserted into the original XML document. This information is defined in the XML DTD with the !ENTITY entry followed by the *name "value"* construction. Afterward, the name defined in the !ENTITY entry is called in the XML document by using the ampersand-prepended name. In Example 10-12, the entity's name is fixed_info, so it is called with &fixed_info.

XSD

XML DTD was the first mechanism to perform validation of XML documents. However, DTD has several drawbacks, including the following:

- It uses non-XML syntax.
- It lacks support for data types.
- It lacks support for namespaces.

To overcome these drawbacks, a new approach to XML validation was created: XML Schema Definition (XSD). Today XSD is used much more often than DTD.

XSD is written in XML, and it follows the XML syntax rules described previously. Because it is used to verify the XML objects, XSD has a predefined set of tags that you can use to create a schema. Example 10-13 shows an XML document from earlier in this chapter, ready to be validated by XSD.

Example 10-13 *XML Document to Be Validated by XSD*

```
<?xml version="1.0" encoding="UTF-8"?>

<vpn xmlns="http://network.programmability/xmldocs/namespace1"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://network.programmability/xmldocs/namespace1 vpn.xsd">

  <customer>NPF_cust1</customer>

  <contact>info@npf.cust1</contact>

  <sites>

    <site id="1">

      <provider_side>

        <router>DUS-1</router>

        <port>Gig1/1/1</port>

        <ipv4>10.0.0.1/30</ipv4>

        <ipv6>fc00::10:0:0:1/126</ipv6>

      </provider_side>

      <customer_side>

        <router>CUST-DUS-1</router>

        <port>xe-0/0/1</port>

        <ipv4>10.0.0.2/30</ipv4>

        <ipv6>fc00::10:0:0:2/126</ipv6>

      </customer_side>

      <enabled/>

    </site>

    <site id='2'>

      <provider_side>

        <router>FRA-1</router>

        <port>1/1/2</port>

        <ipv4>10.0.0.5/30</ipv4>

        <ipv6>fc00::10:0:0:5/126</ipv6>
```

Data Type	Description
String	Any set of characters allowed by XML syntax, enclosed in quotation marks (for example, "info@npf.cust1", "Customer 1")
Decimal	Numeric data, possibly including a decimal point, and indication of whether the value is negative or positive (for example, 1234.45, -12, +123, -0.23)
Integer	Numeric data without a decimal part but with indication of whether the value is negative or positive (for example, 123, -123)
Boolean	Data type associated with Boolean logic, so the allowed values are True or False
Date	Date in the format YYYY-MM-DD (for example, 2021-05-16)
Time	Time of day in the format HH:MM:SS (for example, 14:30:23)

There are many others attributes that you might want to add to an XSD scheme. Two attributes are mandatory for XSD validation:

• **default**: This attribute sets the default value to the validated tag, if there is no value provided.

• **fixed**: This attribute rewrites the value of the validated tag, even if there is no value provided.

Example 10-16 shows how these two attributes would look in an XSD file.

Example 10-16 *Using Optional Attributes in the XSD Element Tag*

```
<xs:element name="customer" type="xs:string" default="Default Customer"/>

<xs:element name="contact" type="xs:string" fixed="do-not-use-this-value"/>
```

Sometimes you need to add stricter validation to an XML document, such as validation against not only a certain data type but also against a possible value. For example, in Example 10-13, you might note that the entry <router> has a hostname of the router where the customer's service is terminated. Despite the fact that the router's hostname is an arbitrary value, you should associate the service with the existing router. Therefore, you need to provide the valid router's hostname in the XML document and make sure that the XML schema can validate that.

In general, an element that has a single tag with a single value is called a *simple object*, and simple objects are typically not mentioned in an XSD document unless you need to impose further limitations, such as content verification. Say that you have a network with four routers, called FRA-1, DUS-1, BLN-1, and MNC-1. In an XSD document, you can add the validation of the <router> entry as shown in Example 10-17.

Example 10-17 *Content Validation in an XSD Document Based on Predefined Values*

```
! Validated entry

<router>FRA-1</router>



! XSD validation

<xs:element name="router">

   <xs:simpleType>

      <xs:restriction base="xs:string">

         <xs:enumeration value="FRA-1"/>

         <xs:enumeration value="DUS-1"/>

         <xs:enumeration value="BLN-1"/>

         <xs:enumeration value="MNC-1"/>

      </xs:restriction>

   </xs:simpleType>

</xs:element>
```

The tag <xs:simpleType> defines that the validated entry is a simple object. The tag <xs:restriction>, which provides the mechanism to verify the tags' values, has a single attribute base that identifies the data type to be verified. You might notice that the attribute isn't called type anymore, as it is in Example 10-16. All the tags nested in <xs:restriction> are validation rules. In Example 10-17, the rule has an enumeration type that is drop-down list of allowed values. All the possible values need to be provided separately; each value has its own entry.

There are multiple options for content validation. One of them, based on the Linux regular expression, is widely used due to its high efficiency. Example 10-18 shows such an option.

Example 10-18 *Content Validation in an XSD Document Using* ***regexp***

```
! Validated entry

<contact>info@npf.cust1</contact>
```

```
! XSD validation
<xs:element name="contact">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:pattern value="([a-zA-Z0-9\.\-\_])+\@([a-zA-Z0-9\.\-\_])+"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

The validation rule in Example 10-18 is called <pattern>, and it verifies whether the entry's value matches the desired regular expression provided in the attribute <value>. In Example 10-18, the desired pattern is a character set containing uppercase and lowercase letters, numbers and characters such as ., -, and _, followed by the @ character and then the same character set. Both before and after the @, there should be at least one character. This is a standard format for mailboxes, and you might want to implement such a check to avoid problems when customer contact information is provided in an incorrect format. You can adapt Example 10-18 to make it more sophisticated and suit your needs.

Although pattern validation is applicable to the numeric data (for example, integer, decimal) as well, you might want to add numeric-specific validation rules. The following two rules are there ones that are most commonly used:

• **minInclusive**: This validation rule sets the lowest possible value for a number (for example, <xs:minInclusive value="0">).

• **maxInclusive**: This validation rule caps the range at the maximum allowed value, including the provided value itself (for example, <xs:maxInclusive value="100">).

Besides validating entry values, you might need to validate entry attributes. To do that, you use another XSD tag, <attribute>, which together with the namespace prefixes results in <xs:attribute>. This is an empty entry tag that is similar to <xs:element>, as you can see in Example 10-19.

Example 10-19 *Attribute Validation in XSD*

```
! Original XML document entries
   <site id='1'>



! Relevant checks in XSD
<xs:attribute name="id" type="xs:integer" use="required"/>
```

The mandatory attribute's name and type within the tag <xs:attribute> are the same as for the tag <xs:element>, but you can see in Example 10-19 one more optional attribute: use. By nature, attributes are not mandatory in an XML document. Therefore, if an attribute is mandatory according to the logic of an application, you should mention that in your XSD schema.

Example 10-19 has one drawback: It doesn't show how the attribute is related to the original tag. The reason is that the tag with the attribute is not a simple object like the ones covered so far. It is a complex object.

There are four types of complex elements in XML schemas:

• Empty elements

• Elements containing only other elements

• Elements containing only some text

• Elements containing mixed information

The first type is shown in Example 10-19, where it provides validation for elements that have only attributes without any textual data. The correct XSD for validation looks as shown in Example 10-20.

Example 10-20 *Complex Element for Attribute Validation in XSD*

```
! Original XML document entries
   <site id="1">



! Relevant checks in XSD
<xs:element name="site">
  <xs:complexType>
    <xs:attribute name="id" type="xs:integer" use="required"/>
  </xs:complexType>
</xs:element>
```

You can see in Example 10-20 that the entry <xs:element> has another nested entry <xs:complexType>, which in turn contains all the validations related to the entry in the XML document. In Example 10-20, it has a single attribute validation, which is the same string as provided in Example 10-19.

The second type of complex object is an element containing other elements. There are two ways such an element can be used: either to add a

<complexType> entry directly to an object or to create a named <complexType> and call it by its name in a validated element. Obviously, the second option is much more scalable. To emphasize this, take a look at the validated object in Example 10-13. You can see that the elements <customer_side> and <provider_side> have the same internal content; you can therefore take advantage of the named objects of <complexType>, as shown in Example 10-21.

Example 10-21 *Complex Type for Element Validation in XSD Using Named Elements*

```
! Original XML document entries

      <provider_side>

        <router>DUS-1</router>

        <port>Gig1/1/1</port>

        <ipv4>10.0.0.1/30</ipv4>

        <ipv6>fc00::10:0:0:1/126</ipv6>

      </provider_side>

      <customer_side>

        <router>CUST-DUS-1</router>

        <port>xe-0/0/1</port>

        <ipv4>10.0.0.2/30</ipv4>

        <ipv6>fc00::10:0:0:2/126</ipv6>

      </customer_side>




! Relevant checks in XSD

<xs:element name="provider_side" type="connectivity_info"/>

<xs:element name="customer_side" type="connectivity_info"/>

<xs:complexType name="connectivity_info">

  <xs:sequence>

    <xs:element name="router" type="xs:string" use="required"/>

    <xs:element name="port" type="xs:string" use="required"/>

    <xs:element name="ipv4" type="xs:string" use="required"/>

    <xs:element name="ipv6" type="xs:string" use="required"/>

  <xs:sequence/>

</xs:complexType>

</xs:element>
```

Example 10-21 includes element <xs:complexType>, which is created as a standalone entry with the attribute name having value "connectivity_info". Inside this element is another nested element, <xs:sequence>, which contains the validated elements provided in the same form as was done earlier for the simple objects. Note that <xs:sequence> requires the validated elements to appear in the defined sequence and not randomly.

Once a named complex object is created, you can use it in the validation of elements that have further nesting. As you can see in Example 10-21, this is done with **type="connectivity_info"** in the elements that need to be validated.

Despite the fact that named complex type elements are very useful, in some cases, you can use ordinary unnamed objects to fulfil the validation requirements. Example 10-22 extends the previous example by adding one more level in the tree.

Example 10-22 *Complex Type for Element Validation in XSD for Mixed Objects*

```
! Original XML document entries

   <site id='1'>

      <provider_side>

        <router>DUS-1</router>

        <port>Gig1/1/1</port>

        <ipv4>10.0.0.1/30</ipv4>

        <ipv6>fc00::10:0:0:1/126</ipv6>

      </provider_side>

      <customer_side>

        <router>CUST-DUS-1</router>

        <port>xe-0/0/1</port>

        <ipv4>10.0.0.2/30</ipv4>

        <ipv6>fc00::10:0:0:2/126</ipv6>

      </customer_side>

   </site>
```

```
! Relevant checks in XSD
<xs:element name="site">
  <xs:complexType>
    <xs:attribute name="id" type="xs:integer" use="required"/>
    <xs:sequence>
      <xs:element name="provider_side" type="connectivity_info"/>
      <xs:element name="customer_side" type="connectivity_info"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType name="connectivity_info">
  <xs:sequence>
    <xs:element name="router" type="xs:string" use="required"/>
    <xs:element name="port" type="xs:string" use="required"/>
    <xs:element name="ipv4" type="xs:string" use="required"/>
    <xs:element name="ipv6" type="xs:string" use="required"/>
  <xs:sequence/>
</xs:complexType>
</xs:element>
```

In Example 10-22, <site> is a parent element for both the <customer_side> and <provider_side> elements. You can create another element, named <xs:complexType>, to validate the site. However, you can also extend the schema created in Example 10-20 by adding the <xs:sequence> element and putting all the child elements that need to be validated directly there. If the order of the elements isn't important, you can use <xs:all> instead of <xs:sequence> to contain all the nested checks.

In certain circumstances, a validated element might have logic that it contains either one or another nested element but not both simultaneously. Say that <site> should contain either the <provider_side> or <customer_side> element. In this case, the schema's element <xs:sequence> should be replaced with the <xs:choice>, as illustrated in Example 10-23.

Example 10-23 *Complex Type for Element Validation in XSD for Mixed Objects*

```
! Relevant checks in XSD
<xs:element name="site">
  <xs:complexType>
    <xs:attribute name="id" type="xs:integer" use="required"/>
    <xs:choice>
      <xs:element name="provider_side" type="connectivity_info"/>
      <xs:element name="customer_side" type="connectivity_info"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
! Further output is truncated for brevity
```

By this point, you should have an understanding of the XML Schema Definition (XSD), including the overall structure and details of building the validation entries.

Brief Comparison of XSD and DTD

Table 10-3 highlights the differences between DTD and XSD. As you can see from this comparison, when developing XML-based data structures today, you should use XSD.

Table 10-3 *DTD and XSD Comparison*

DTD	XSD
DTD syntax is different from that of XML.	XSD is written in XML, so there is no need to learn a separate language.
Data types aren't defined in DTD.	XSD defines the data types for elements.
There is no concept of namespaces in DTD.	As it is written in XML, XSD natively supports XML namespaces.
DTD is not extensible.	XSD is extensible, using standard XML.

Navigating XML Documents

So far in this chapter, you have learned about various topics related to XML documents and schemas. In this section of the chapter, you will learn how to navigate an XML document and extract the information that is necessary for XML document transformation. The transformation and associated techniques are covered later in this chapter, in the section "XML Stylesheet Language Transformations (XSLT)."

XPath

Recall that each XML document starts with at least one root element, and there may be many parent/child relationships in the hierarchy. Therefore, an XML document could be treated as a tree, where each leaf is a piece of the information that can be accessed along some branches. XPath is a key element in the navigating the XML tree and choosing the necessary data. XPath uses path expressions, which are similar to the path format used in Linux. However, XPath also has plenty of unique features.

The primary XPath building block is a node. There are seven node types defined for XPath (https://www.w3schools.com/xml/xpath_nodes.asp):

• Element

• Attribute

• Text

• Namespace

• Processing instruction

• Comment

• Document

From an XPath perspective, every XML document is a tree of nodes that starts from the root element. The connectivity between the nodes is defined by the various types of relationships. The relationships between various nodes are similar to the relationships that exist in XML documents:

• **Parent**: Each node besides the root element has a parent node. Each node can have only one parent node.

• **Children**: Each node, including the root element, may have one or more children.

• **Siblings**: All the children of the same parent are the siblings to each other.

• **Ancestors**: All the parent nodes up to the root element (for example, parent, parent's parent) are ancestors.

• **Descendants**: All the nested children down to the all leafs (for example, children, children's children) are descendants.

Table 10-4 lists and describes the XPath syntax elements.

Table 10-4 *XPath Syntax*

Expression	Description
nodename	Selects all nodes with the name *nodename*
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection, no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes
[]	Predicate element, which adds the details to the original path to make the node's choice more precise

Example 10-24 shows the XML document we use in this section to examine XPath.

Example 10-24 *XML Document for XPath Expressions*

```
<?xml version="1.0" encoding="UTF-8"?>
<vpn>
  <customer>NPF_cust1</customer>
  <contact>info@npf.cust1</contact>
  <sites>
    <site id="1">
      <provider_side>
        <router>DUS-1</router>
        <port>Gig1/1/1</port>
      </provider_side>
      <customer_side>
        <router>CUST-DUS-1</router>
        <port>xe-0/0/1</port>
      </customer_side>
```

```
      <enabled/>
    </site>
    <site id="2">
      <provider_side>
        <router>FRA-1</router>
        <port>1/1/2</port>
      </provider_side>
      <customer_side>
        <router>CUST-FRA-1</router>
        <port>Ethernet1</port>
      </customer_side>
      <enabled/>
    </site>
  </sites>
</vpn>
```

As you see, Example 10-24 provides a simplified version of a XML document shown earlier in this chapter (refer to Example 10-13).

XPath itself doesn't extract any data, as it is just a way to define the path. Therefore, XPath is widely used in XSLT, JavaScript, Python, and so on. If XPath is used with a tool, you see which element will be collected in relationship to a specific path. Example 10-25 shows a simple path expression that uses an absolute path to a node.

Example 10-25 *Simple XPath Expression with an Absolute Path*

```
Path:   /vpn/customer

Result: NPF_cust1
```

The requested path is associated with an element, and it starts from the root element vpn and continues with its child's node customer. The node contains text that is returned as a result. You can achieve the same result by using another type of selection, as shown in Example 10-26. This type of selection with the XPath syntax means that all the nodes with the name customer will be selected, regardless of where in the tree they appear.

Example 10-26 *Simple XPath Expression with an "anywhere" Selection*

```
Path:   //customer

Result: NPF_cust1
```

The end result might be less obvious if you have a list of elements with the same nodes inside, as shown in Example 10-27.

Example 10-27 *Simple XPath Expression with an Absolute Path and Multiple Outputs*

```
Path:   /vpn/sites/site/provider_side/router

Result: DUS-1

        FRA-1
```

The path expression provided doesn't specify any particular element in the list. That's why, if there are multiple elements having the same name in the XML tree, all the results will be provided. To be more specific, you can use predicates. Example 10-28 illustrates such an approach.

Example 10-28 *XPath Expression with a Predicate*

```
Path:   /vpn/sites/site[1]/provider_side/router

Result: DUS-1
```

According to the XML standard, the numbering of the elements starts with the index 1 (which is very different from Bash or Python). Therefore, the predicate [1] applied to the node site in the XPath expression indicates that the first site element is chosen from the list. However, filtering based on the element's position is not a very clean solution, as you need to know the indexing details. To overcome this, you can base the element's choice on an attribute, as shown in Example 10-29.

Example 10-29 *XPath Expression with a Predicate and an Attribute*

```
Path:   /vpn/sites/site[@id="2"]/provider_side/router

Result: FRA-1
```

In the path provided inside the predicate, the attribute search is used. Per Table 10-4, the attribute's search starts with @ followed by the name of the attribute's logical operator and the value that the attribute is compared against. The logical operator may be any of the expressions listed in Table 10-5.

Table 10-5 *XPath Logical Operator Values*

Operator	Description
=	The values are equal. This operator works for both textual and numeric data.
!=	The values are not equal. This operator works for both textual and numeric data.
>	The attribute's value is more than the value it is compared to.
<	The attribute's value is less than the value it is compared to.

You can also combine the different methods to define a path. Example 10-30 shows how you can get information about all the routers in the network.

Example 10-30 *XPath Expression with Two Methods of Path Definition*

```
Path:   /vpn/sites//router

Result: DUS-1

        CUST-DUS-1

        FRA-1

        CUST-FRA-2
```

So far you have learned how you can navigate an XML document by using XPath for flexible information choice. You will learn more about XPath later in this chapter, in examples of working with Python.

XML Stylesheet Language Transformations (XSLT)

Recall that XML is a language you can use to create data structures. However, sometimes you might need to create a sort of custom XML file based on another file or even based on multiple XML files. You can achieve this by using XSLT. XSLT is a language used to create other documents (for example, HTML, XML) based on XML input. XSLT relies heavily on XPath to collect relevant data.

The following are some of the main elements of XSLT:

• **<template>**: This element is used to create a template that transforms the initial XML document into a new format.

• **<value-of>**: This element points to a particular piece of information in the original XML document by using XPath.

• **<for-each>**: This element creates a loop over a certain set of the objects (for example, a list) and uses XPath to address it.

• **<if>**: This element settles a condition which implies that the child's elements will work only if the condition is true.

The best way to show the application of XSLT is with an example. Example 10-31 shows a basic XSLT stylesheet.

Example 10-31 *XSLT Stylesheet*

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">

    <html>

      <body>

        <table>

          <tr>

            <th>Customer</th>

            <th>Contact</th>

          </tr>

          <xsl:for-each select="vpn">

            <tr>

              <td><xsl:value-of select="customer"/></td>

              <td><xsl:value-of select="contact"/></td>

            </tr>

          </xsl:for-each>

        </table>

      </body>

    </html>

  </xsl:template>

</xsl:stylesheet>
```

The first thing you might notice in Example 10-31 is that the document starts with the XML declaration, in much the same way that all other XML documents in this chapter start. Then you see that the root element is <stylesheet>, which is a standard for the XSLT. This element has an attribute assigned that contains the namespace of the XSLT dictionary associated with the prefix xsl. As explained earlier in this chapter, this prefix prepends all the elements associated with this namespace.

The element <xsl:template> defines what will be the result of the input XML transformation. It also has the attribute match, which identifies the part of the input document to which the transformation shall be applied. In Example 10-31, the value of the attribute is /, which means the transformation will be applied to the root of the target XML document.

Inside the <xsl:template> element is a mix of HTML tags (as this template generates an HTML document) and the XSLT elements described earlier in this document. Each of the XSLT elements has the attribute select, which contains the XPath element to certain data in the target XML document.

The exact process an application goes through with XSLT transformation is beyond the scope of this chapter, as it might be created in any Web framework (for example, PHP, JavaScript). However, based on the transformation of the XML document from Example 10-3, using the XSLT stylesheet from Example 10-31 results in the document shown in Example 10-32.

Example 10-32 *Result of the XSL Transformation*

```
Customer      Contact

NPF_CUST1  info@npf.cust1
```

Processing XML Files with Python

This section provides examples of processing and modifying the XML document from Example 10-3. That XML document is stored in the file input.xml, located in a directory together with the file xml_processing.py, which contains the Python code shown in Example 10-33.

Example 10-33 *Working Folder for a Python Script for XML Processing*

```
$ ls -l

total 16

-rw-r--r--  1 npf_user  npf_user  960 11 Feb 21:19 input.xml

-rwxr-xr-x  1 npf_user  npf_user  244 11 Feb 21:54 xml_processing.py
```

Note

Python is covered in detail in Chapters 5, "Python Fundamentals," and 6, "Python Applications," so if you need a refresher on working with Python modules and functions, refer to those chapters.

There is a module in a standard Python 3.x package used for XML processing called **xml.etree.ElementTree**. According to the official documentation at https://docs.python.org, this module implements a simple and efficient API for parsing and creating XML data. Using this module, you can create an object by parsing the XML file read from certain paths, as shown in Example 10-34.

Example 10-34 *Creating a Python Script for XML Processing*

```
$ cat xml_processing.py

#!/usr/bin/env python


# Modules

import xml.etree.ElementTree as ET


# Body

tree = ET.parse('input.xml')

root = tree.getroot()
```

In Example 10-34 the **xml.etree.ElementTree** module for XML processing is imported in the Python script (under the name ET to simplify its usage later in the code). Next, the object **tree** is created using the method **parse** of the **ET** module from the file **input.xml**. After that, the object **root** is created from the **tree** using the **getroot** method of the element's object. The **root** object contains the whole tree from the XML document, which can be used in various shapes or forms using further methods or properties of this module. Table 10-6 highlights the most important of these methods or properties for your reference.

Table 10-6 ***xml.etree.ElementTree*** *Properties and Methods*

Property/Method		Description
tag property		Returns the name of an XML element
text property		Returns the value of a certain XML element
attrib property		Returns the attributes associated with a certain XML element
getroot() method		Returns the data tree starting from the root element of the XML document
findall() method		Returns the list of the objects associated with a certain path, provided in XPath format
find() method		Returns the first matched object associated with a certain path, provided in XPath format
get() method		Returns the value of an attribute associated with a certain XML element
set() method		Modifies an attribute's value associated with a certain XML element
append() method		Adds a sub-element at the end of the existing XML document
write() method		Saves the Python object with the XML data into a file

Let's look at an example that demonstrates how to read data from the XML document and modify it. The first operation is to get the data, as

shown in Example 10-35.

Example 10-35 *Extracting Element Names and Attributes from an XML Element (Python Code and Code Execution)*

```
$ cat xml_processing.py
#!/usr/bin/env python

# Modules
import xml.etree.ElementTree as ET

# Body
tree = ET.parse('input.xml')
root = tree.getroot()

print('Building catalog for {} services:'.format(root.tag))

for site_entry in root.findall('.//sites/'):
    site_id = site_entry.get('id')

    print('{} {}'.format(site_entry.tag, site_id))



$ ./xml_processing.py
Building catalog for vpn services:
site 1
site 2
```

In Example 10-35, you can see that the Python code generates some text that includes information from the initial XML document. The first **print()** function has simple input consisting of fixed text and the name of the root element provided by the **tag** property of the **root** object.

The next part of the code is a **for** loop. Based on the information provided in Table 10-6, you know that the method **findall()** returns a list of the sub-elements on the provided XPath. The XPath in this case is ***//sites/***, which refers to the child objects of the parent element **<sites>**, which can be located anywhere in the root tree. If you examine the original XML document from Example 10-3, you might find that there are two **<site>** elements, which are siblings to each other. By using the **get()** method to search for the value of the **id** attribute, this value is extracted and assigned to the variable **site_id**. The function **print()** within the loop prints this **site_id** value together with the name of topmost tag of this sub-element returned using the **tag** property of the **site_entry** object.

Example 10-36 shows how to extract the values of the elements using Python.

Example 10-36 *Extracting Values from an XML Element*

```
$ cat xml_processing.py
#!/usr/bin/env python

# Modules
import xml.etree.ElementTree as ET

# Body
tree = ET.parse('input.xml')
root = tree.getroot()

print('Building catalog for {} services:'.format(root.tag))

for site_entry in root.findall('.//sites/'):
    site_id = site_entry.get('id')

    print('{} {}'.format(site_entry.tag, site_id))

    router_a = site_entry.find('./provider_side/router').text
    port_a = site_entry.find('./provider_side/port').text
    router_b = site_entry.find('./customer_side/router').text
    port_b = site_entry.find('./customer_side/port').text
```

```
    print('Connectivity: {} // {} <---> {} // {}'.format(router_a, port_a, router_b,
port_b))
```

```
$ ./xml_processing.py
Building catalog for vpn services:
site 1
Connectivity: DUS-1 // Gig1/1/1 <---> CUST-DUS-1 // xe-0/0/1
site 2
Connectivity: FRA-1 // 1/1/2 <---> CUST-FRA-1 // Ethernet1
```

To get the element's value, you call the **text** property of the object. In Example 10-36, you can see that the object is a sub-element created using the **find()** method with the associated XPath. In contrast to **findall()**, which returns a list regardless of the number of entries in the list, the **find()** method always returns a single object. When you know that an element doesn't have any further children, you can safely use **find()**to collect end values. Then you can use the **text** property to get a particular value.

When the script in Example 10-36 is executed, the relevant values are collected. The report that is generated shows how the network elements are connected to each other, based on information from the XML document used.

At this point, you have learned about all the major cases and how to extract data from an XML document. The only topic left to cover is modification of an XML document.

To provide some useful context, say that you need to add to an XML document some information about a third site. There are multiple ways to provide the information about the new site. For instance, in Example 10-37, the data about the new site is provided as a Python dictionary.

Example 10-37 *A Python Dictionary with Information About a New Site*

```
$ cat xml_modification.py
#!/usr/bin/env python

# Modules
import xml.etree.ElementTree as ET

# Variables
site_data = {
                "3": {
                    "provider_side": {
                        "router": "BLN-1",
                        "port": "swp1",
                        "ipv4": "10.0.0.8/30",
                        "ipv6": "fc00::10:0:0:8/126"
                    },
                    "customer_side": {
                        "router": "CUST-BLN-1",
                        "port": "Gig1",
                        "ipv4": "10.0.0.9/30",
                        "ipv6": "fc00::10:0:0:9/126"
                    }
                }
            }

# Body
```

Note

For this example, the new file xml_modification.py is created in the same folder where the previous script is located.

To create the XML element, you can use two new functions of the **ET** module: **Element()** and **SubElement()**. **Element()** creates the root XML object, and **SubElement()** adds the child's element to the parent. Example 10-38 demonstrates the process of generating the XML element out of the Python dictionary.

Example 10-38 *Converting a Python Dictionary to an XML Element*

```
$ cat xml_modification.py
#!/usr/bin/env python
```

```
# Modules
import xml.etree.ElementTree as ET


# Variables
site_data = {
                "3": {
                    "provider_side": {
                        "router": "BLN-1",
                        "port": "swp1",
                        "ipv4": "10.0.0.8/30",
                        "ipv6": "fc00::10:0:0:8/126"
                    },
                    "customer_side": {
                        "router": "CUST-BLN-1",
                        "port": "Gig1",
                        "ipv4": "10.0.0.9/30",
                        "ipv6": "fc00::10:0:0:9/126"
                    }
                }
            }


# Body
for l1_key, l1_var in site_data.items():
    XML_element_L1 = ET.Element('site')
    XML_element_L1.set('id', l1_key)

    for l2_key, l2_var in l1_var.items():
        XML_element_L2 = ET.SubElement(XML_elemnt_L1, l2_key)

        for l3_key, l3_var in l2_var.items():
            XML_element_L3 = ET.SubElement(XML_elemnt_L2, l3_key)
            XML_element_L3.text = l3_var


ET.dump(XML_element_L1)
```

```
./xml_modification.py
<site id="3"><provider_side><router>BLN-
1</router><port>swp1</port><ipv4>10.0.0.8/30</ipv4><ipv6>fc00::10:0:0:8/126</ipv6></pro
vider_side><customer_side><router>CUST-BLN-
1</router><port>Gig1</port><ipv4>10.0.0.9/30</ipv4><ipv6>fc00::10:0:0:9/126</ipv6></cus
tomer_side></site>
```

The conversion of the Python dictionary to the XML element is done in a bunch of nested **for** loops created over the Python dictionary using the **items()** function. Thanks to this function, you don't need to think about the elements' names during the script creation; rather, you can offload this logic to the dictionary itself, as the dictionary key names and values are used to create both the XML element tags and values.

Then, within the loops, you can see that an XML element (root) is created using the **XML_element_L1** variable, which is the result of the **ET.Element()** function's execution. The root element tag is **site**. Then, using the **set()** method, you can set the attribute to any element.

All the XML elements nested under the root element are added using the **ET.SubElement()** function, which has two arguments for input: the name of the parent's element (e.g., **XML_elem_L2**) and the name of the element to be created (e.g., **l2_key**). Finally, to add the value to any XML element, you need to add it to the property **text** of the XML object.

All software development, including XML creation, requires careful debugging. The **ET** module has the function **dump()**, which allows you to print the output of a whole XML element, including all the levels of the hierarchy. The input to the function is, logically, the XML element itself. Therefore, the outcome of the **xml_modification.py** execution is a printed XML document.

The final stage of this journey involves integrating the created XML element into the original XML element that contains all the sites. Example 10-39 shows the full script for this final task.

Example 10-39 *Adding One XML Element to Another*

```
$ cat xml_modification.py

#!/usr/bin/env python

#

# THE OUTPUT IS TRUNCATED FOR BREVITY

#

# Body

for l1_key, l1_var in site_data.items():

    XML_element_L1 = ET.Element('site')

    XML_element_L1.set('id', l1_key)


    for l2_key, l2_var in l1_var.items():

        XML_element_L2 = ET.SubElement(XML_element_L1, l2_key)


        for l3_key, l3_var in l2_var.items():

            XML_element_L3 = ET.SubElement(XML_element_L2, l3_key)

            XML_element_L3.text = l3_var


tree = ET.parse('input.xml')

root = tree.getroot()


print('Adding elements to {} services:'.format(root.tag))


root.find('.//sites').append(XML_element_L1)

tree.write('output.xml')



$./xml_modification.py
```

Example 10-39 shows how to add one of the XML elements to another by using the **append()** method, with the new XML element provided as an input to this method and the method itself applied to the original XML element. If you need to add the new XML element to a specific path of the original element, you can do so by using the **find()** method with the appropriate XPath, as shown in Example 10-39.

After the element is modified, you can either verify it by using the **ET.dump()** function or save it to another file by using the **write()** method. Example 10-39 uses the latter approach, so you don't see anything in the console output when the Python script executes.

If you verify the file output.xml that is generated by this script, you see the output shown in Example 10-40.

Example 10-40 *The XML Element After a Merger with Another One*

```
$ cat output.xml

<vpn>
  <customer>NPF_cust1</customer>
  <contact>info@npf.cust1</contact>
  <sites>
    <site id="1">
      <provider_side>
        <router>DUS-1</router>
        <port>Gig1/1/1</port>
        <ipv4>10.0.0.1/30</ipv4>
        <ipv6>fc00::10:0:0:1/126</ipv6>
      </provider_side>
      <customer_side>
        <router>CUST-DUS-1</router>
        <port>xe-0/0/1</port>
        <ipv4>10.0.0.2/30</ipv4>
        <ipv6>fc00::10:0:0:2/126</ipv6>
      </customer_side>
      <enabled />
```

```
    </site>
    <site id="2">
      <provider_side>
        <router>FRA-1</router>
        <port>1/1/2</port>
        <ipv4>10.0.0.5/30</ipv4>
        <ipv6>fc00::10:0:0:5/126</ipv6>
      </provider_side>
      <customer_side>
        <router>CUST-FRA-1</router>
        <port>Ethernet1</port>
        <ipv4>10.0.0.6/30</ipv4>
        <ipv6>fc00::10:0:0:6/126</ipv6>
      </customer_side>
      <enabled />
    </site>
  <site id="3"><provider_side><router>BLN-
1</router><port>swp1</port><ipv4>10.0.0.8/30</ipv4><ipv6>fc00::10:0:0:8/126</ipv6></pro
vider_side><customer_side><router>CUST-BLN-
1</router><port>Gig1</port><ipv4>10.0.0.9/30</ipv4><ipv6>fc00::10:0:0:9/126</ipv6></cus
tomer_side></site></sites>
</vpn>
```

The indentations in Example 10-40 do not look quite right, but the opening and closing tags are correct, and you can see that the new element **<site id="3">** has been added exactly where it should be: right after closing of the previous **</site>** sibling. From an XML processing point of view, the XML document is generated correctly.

To verify that the document, you can run the script from Example 10-36 but change the input XML file from input.xml to output.xml, as shown in Example 10-41.

Example 10-41 *Rerunning XML Processing for the Updated XML Document*

```
$ cat xml_processing.py
!
! OUTPUT IS TRUNCATED FOR BREVITY
!
tree = ET.parse('input.xml')
! FURTHER OUTPUT IS TRUNCATED FOR BREVITY
!



$ ./xml_processing.py
Building catalog for vpn services:
site 1
Connectivity: DUS-1 // Gig1/1/1 <---> CUST-DUS-1 // xe-0/0/1
site 2
Connectivity: FRA-1 // 1/1/2 <---> CUST-FRA-1 // Ethernet1
site 3
Connectivity: BLN-1 // swp1 <---> CUST-BLN-1 // Gig1
```

The output of the XML processing for the newly created site is consistent with the previous elements, which confirms that the XML modification was successful.

Summary

This chapter covers the following points:

- XML is a markup language for creating data structures, and it was the first language created for this purpose.
- Generally, in contrast to HTML, XML has an arbitrary syntax, meaning that there are no predefined names for the tags.
- XML has some strict rules, such as sequencing of the opening/closing tags and beginning a document with an XML declaration.
- XML has additional (optional) metadata in the form of attributes associated with the XML elements' tags.

• XML doesn't include information about the data type inside the XML messages, so additional mechanisms are required to verify its validity.

• XML validation techniques are based on DTD and XSD schemas.

• XSD is more flexible than DTD, and it is written in XML.

• An XML namespace provides a mapping between an XML document and an XSD file.

• To address a specific part of an XML document, you can use special links based on XPath addresses.

• XSLT can transform an initial XML document into any other document, including another XML document, an HTML document, or a document of any other format.

• The Python library **xml.etree.ElementTree** allows you to deal with XML elements, including parsing an XML document, extracting the data from the document, modifying an element, and saving elements.

Chapter 11. JSON and JSD

This chapter covers the encoding format JavaScript Object Notation (JSON). Much like XML, JSON is used to encode the messages of network programmability protocols. Whereas NETCONF supports only XML, RESTCONF and gRPC both support JSON. JSON is newer than XML. Whereas XML was primarily developed for machine-to-machine communications, JSON was developed to be human readable. JSON therefore tends to be the more popular choice for encoding when specific XML features are not required.

This chapter also covers one of the applications of JSON: JSON Schema Definition (JSD). JSD is used to construct schemas, or data models, and can be used either independently or in conjunction YANG (which is covered in detail in Chapter 13, "YANG").

JavaScript Object Notation (JSON)

A number of RFCs define JSON, but the main one is RFC 8259, "The JavaScript Object Notation (JSON) Data Interchange Format." This RFC provides an accurate and straightforward description of what JSON is. According to this RFC, JSON is a "lightweight, text-based, language-independent data interchange format." These three characteristics all contribute to the success of the JSON:

• **Lightweight**: The structure of JSON is straightforward, and it is easy to start using JSON.

• **Text based**: You can create data in the JSON format by using any kind of text editor, as it doesn't require any specific software or application.

• **Language independent**: The vast majority of programming and scripting languages today support data in the JSON format. This is particularly important because it allows applications written in different languages to easily interoperate with each other.

Due to its capabilities, JSON is now one of the most critical and widely used data formats. For example, JSON is the number-one data structure format for managing applications through REST APIs and in RESTCONF as well, as it has a very logical and straightforward structure. Example 11-1 provides an example.

Example 11-1 *JSON Data Example*

```
{
  "book_title": "Network Programmability and Automation, Volume I",
  "publisher": "Cisco Press",
  "pages": 925,
  "authors": ["Jeff Doyle" , "Khaled Abuelenain", "Ahmed Elbornou", "Anton Karneliuk"]
}
```

Example 11-1 provides a brief description of this book in JSON format. This data is easily readable and can be processed by any application (for example, Ansible) or programming language (for example, Python). The following pages describe the JSON data format.

JSON Data Format and Data Types

As already mentioned in this chapter, JSON is a data format that shows how data is stored and represented. Each JSON object is a set of key/value pairs that contains information relevant to a particular application. Example 11-2 shows a simple JSON object with a single key/value pair.

Example 11-2 *A Simple JSON Object*

```
{
  "book_title": "Network Programmability and Automation, Volume I"
}
```

A JSON object is always framed with the symbol **{**at the beginning and the symbol **}** in the end. All the key/value pairs are contained within this framing. Example 11-2 shows a single pair, where "book_title" is a key, and "Network Programmability and Automation, Volume I" is the value.

When a JSON object contains more entries than a single key/value pair, the entries are divided by commas, as illustrated in Example 11-3. The comma separator is essential because the absence of the comma triggers error in all the applications, which means the data isn't processed.

Example 11-3 *Multiple Key/Pair Values Inside a JSON Object*

```
{
  "book_title": "Network Programmability and Automation, Volume I",
  "pages": 925
}
```

As you can see in the examples shown so far, the key is always framed with quotation marks. However, the value isn't always framed with quotation marks; whether it is depends on the type of the value. Table 11-1 lists the six data types defined within JSON.

Table 11-1 *The JSON Data Types*

Type		Description	Example
String		Any textual data, which is processed as a string.	"some value"
Number		Any numeric data in decimal format, including all math actions.	123
Boolean		Boolean data with just two possible values: true or false.	true/false
Null		An empty value that is used when you need to have a key even if there is no value associated.	null
Object		A value for a key that is framed with {} symbols.	{"a": "b"}
Array		A value for a key that is a list containing entries in any other format. The elements of an array are separated by commas and are framed with [] symbols.	["c", 12, true, {"e": "f"}]

You can see in Table 11-1 that four of the basic data types provide a single value to a key. The other two data types contain more than a single value and, if necessary, make it possible to create a complex hierarchical data structure.

Example 11-4 puts all the JSON data types in context so that you can better understand them.

Example 11-4 *Using All the JSON Data Types Together*

```
{
  "book_title": "Network Programmability and Automation, Volume I",
  "publisher": "Cisco Press",
  "pages": 925,
  "published": false,
  "release_date": null,
  "authors": ["Jeff Doyle" , "Khaled Abuelenain", "Ahmed Elbornou", "Anton Karneliuk"],
  "sample_content": {
    "introduction": "This is an awesome book about network programmability and automation!",
    "chapters": [
      {
        "name": "Chapter 1",
        "description": "The Network Programmability and Automation Ecosystem"
      },
      {
        "name": "Chapter 2",
        "description": "Linux Fundamentals"
      }
    ]
  }
}
```

In the snippet shown in Example 11-4, you can see all the data types together:

• **"book_title" and "publisher":** String data type framed in quotation marks

• **"pages":** Number data type provided as just a number without quotation marks

• **"published":** Boolean data type

• **"release_date":** Null data type (because the release data isn't yet available at the time of writing)

• **"authors":** Array (or list) data type that contains the values in string data format

• **"sample_content":** JSON object nested as a value

• **"introduction" under "sample_content":** Ordinary key/value with string format

• **"chapters" under "sample_content":** Array (or list) data type that contains JSON objects as elements

As long as you follow the rules of the JSON notation, there are no boundaries in terms of how you structure your data aside from those that might be imposed by the application you are dealing with.

Note

The quotation marks in the framing value are significant, and you need to be very careful with them. For example, the value **10** is a numeric data, and you can apply all math operations to it; on the other hand, **"10"** is a string value, and the rules of string processing are applied to it. Booleans and strings also require careful attention to quotation marks, as the value **true** is a Boolean type and isn't equal to **"true"**, which is a string type. The same rule applies to **null** and **"null"**. When you have a series of operations for data processing, it's particularly important to get the quotation marks right in order to get the proper result.

This short section provides all the information you need to know about the JSON data structure and types. JSON is very popular today due to the simplicity and flexibility you have already seen.

JSON Schema Definition (JSD)

Now that you know what the JSON data format is and how you can use it, the next step is to understand one of the immediate applications of JSON: JSON Schema Definition (JSD). As you will learn soon, the YANG language provides almost endless possibilities for creating data models that can be used anywhere, including in network programming. On the other hand, YANG format isn't used for representation or transmission of the actual data. There are specific reasons for that, as you are about to learn.

One of the reasons YANG format isn't used for representation or transmission of data is that presenting real information in YANG format takes a lot of space. A YANG module (or model) provides a detailed definition of all the data types used, their parameters, and so on. If both the transmitter and receiver of the information have the same YANG modules, it isn't necessary to provide all the information about data types within the message itself as doing so would be excessive. It's enough to send only the key/value pairs, which are checked for compliance using the YANG-based dictionary on the receiver side. If JSON format is used to convey those key/value pairs between applications, the JSON schema compiled from YANG modules is a perfect candidate for that "something." Using YANG directly also saves you the effort of converting data from JSON to YANG.

Based on the factors just described, YANG is used to construct data models but not to represent the real information transmitted between different network functions or applications. For such a task, other ways of representing data are used: mainly JSON schemas (covered in this chapter) and XML schemas (covered in Chapter 10, "Extensible Markup Language (XML) and XML Schema Definition (XSD)"). JSON schemas and XML schemas are equally important; whereas JSON data representation is used in RESTCONF protocols, XML data representation is used in NETCONF protocols.

Earlier in this chapter, you learned about the JSON language in general, including its components and how data is represented. In this section, you will learn how JSON schemas are structured and how to create schemas from YANG modules.

Structure of the JSON Schema

The JSON schema is a representation of a data model in a specific format encoded in JSON. Currently, it exists as an IETF draft (although it is already in its seventh version) titled "JSON Schema: A Media Type for Describing JSON Documents." Because it is a draft, as the draft progresses toward becoming RFC, some changes might be made compared to what you read in this section.

The core idea of the JSON schema is to create a clear understanding of data that is transmitted/received in JSON format. The JSON schema tries to answer questions such as these:

• What specification does the schema follow?

• What should be contained in the data model?

• What key/value pairs are mandatory? What key/value pairs are not mandatory?

• What is the data type for a specific value?

• Is the value in the allowed range?

The YANG modules seek to answers the same questions, but JSON mainly focuses on data representation.

To give you a better understanding of how the JSON schema is structured, this section focuses on building one JSON schema, step by step. For this example, say that you need to create a JSON schema for customer IP VPN service. This schema will be used by an SDN controller; because the schema is created for learning purposes, it is not directly implemented in any commercial product. Example 11-5 provides a

starting point for this schema.

Example 11-5 *Basic JSON Schema Without Content*

```
{

  "$schema": "http://json-schema.org/draft-07/schema#",

  "$id": "http://network.vendor/ip-vpn.schema.json",

  "title": "IP-VPN",

  "description": "This is an arbitrary example to show you structure of the JSON schema based on example of IP VPN
service, how it can be implemented in SDN controller",

  "type": "object"

}
```

There are three types of data in Example 11-5:

• **Schema keywords**: A schema keyword starts with the symbol **$**. In Example 11-5, there are two such keywords: **"$schema"** tells JSON which version of the JSON schema should be used for validation. that the time of writing, the current version of the JSON schema is draft-handrews-json-schema-02, and this is the path that is encoded. The second schema keyword is **"$id"**, and it contains the URI for the schema.

• **Schema annotation**: The schema annotation is information attached to the schema for application use. As you have already seen, the schema keywords provide some important values that any application using this JSON schema needs (that is, which JSON schema version to use and what the URI looks like), but there is no information about any content of the schema in the schema keywords. Annotations provide such information. In Example 11-5, **"title"** and **"description"** are schema annotations. Annotations are not mandatory, as they don't take part in the validation. However, using them is a good practice, and it is recommended that you use them.

• **Validation keywords**: The validation keywords add constraints (such as what data type should be used) to the schema that are used later on for the validation process. The **"type"** keyword could potentially be any JSON data type. However, the most widely used data type is the object data type. In Example 11-5, the keyword **"object"**, which is the value of the key **"type"**, indicates that the schema is used for validation of JSON objects.

Now we are ready to talk about what is needed to configure the IP VPN for the customer. IP VPNs are popular types of the VPN services deployed in the Service Providers, which provide IP-based connectivity between a customer's router and an MPLS-based network. Example 11-6 extends the initial JSON schema with some new content for the IP VPN.

Example 11-6 *Properties in the JSON Schema*

```
{

  "$schema": "http://json-schema.org/draft-07/schema#",

  "$id": "http://network.vendor/ip-vpn.schema.json",

  "title": "IP-VPN",

  "description": "This is an arbitrary example to show you structure of the JSON schema based on example of IP VPN
service, how it can be realized in SDN controller",

  "type": "object",

  "properties": {

    "ContractId": {

      "description": "This field contains customer contract ID to track customer services within configuration",

      "type": "integer",

      "exclusiveMinimum": 0

    },

    "CustomerName": {

      "description": "This field contains the customer name to add meta information to configuration, where applicable",

      "type": "string"

    },

    "CustomerContact": {

      "description": "This field contains e-mail address of the customer, which is used for any communication",

      "type": "string"

    },

    "Tags": {

      "description": "This field contains tags associated with the customer",

      "type": "array",

      "items": {
```

```
        "type": "string"
      }
    }
  },
  "required": [ "ContractId", "CustomerName", "CustomerContact" ]
}
```

As you can see, Example 11-6 includes new entries in the JSON schema. The key **"properties"** is called a *validation keyword* because it contains information about the content of expected JSON objects; you can think about it as a RESTCONF message received by an SDN controller. Inside **"properties"** are a variety of keys, each of which is accompanied by the schema annotation **"description"** and the validation keyword **"type"**. As discussed earlier in this chapter, there are six data types available in JSON, and the field **"type"** uses some of those types: a string, an integer, and an array. For the key **"ContractId"**, there is additional validation keyword, **"exclusiveMinimum"**, which imposes constraints so that the value of this key can't be zero. You can find the full list of all the validation keywords in the JSON IETF draft: https://tools.ietf.org/html/draft-handrews-json-schema-validation-01.

Note

SDN controllers are discussed in further detail in Chapter 16, "Service Provider Programmability." For now, you can think of an SDN controller as an application that manages your network functions.

At the end of the schema is a new validation keyword, **"required"**, which contains an array of strings with the names of the keys. These keys are mandatory, which means these values must be present in the received JSON object, or the validation won't be successful. As you can see, not all the keys are listed with **"required"**. It is possible to have additional information that might or might not be presented in the JSON object.

Repetitive Objects in the JSON Schema

The JSON schema in our example is being slowly filled in with application-oriented data, and by now you should be familiar with the basics. However, there not yet any information about the technical details associated with the IP VPN service for which you are building the JSON schema. The next step is to create information about endpoints where the customer IP VPN service is terminated. Example 11-7 shows this information.

Example 11-7 *Definitions in the JSON Schema*

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://network.vendor/ip-vpn.schema.json",
  "title": "IP-VPN",
  "description": "This is an arbitrary example to show you structure of the JSON schema based on example of IP VPN
service, how it can be realized in SDN controller",
  "type": "object",
  "properties": {
    "ContractId": {
      "description": "This field contains customer contract ID to track customer services within configuration",
      "type": "integer",
      "exclusiveMinimum": 0
    },
    "CustomerName": {
      "description": "This field contains the customer name to add meta information to configuration, where applicable",
      "type": "string"
    },
    "CustomerContact": {
      "description": "This field contains e-mail address of the customer, which is used for any communication",
      "type": "string"
    },
    "Tags": {
      "description": "This field contains tags associated with the customer",
      "type": "array",
      "items": {
        "type": "string"
```

```
    }
  },
  "VPNSites": {
    "description": "This array contains objects describing VPN endpoints for the customer service",
    "type": "array",
    "items": {
      "$ref": "#/definitions/VPNEndpoint"
    }
  }
},
"required": [ "ContractId", "CustomerName", "CustomerContact", "VPNSites" ],
"definitions": {
  "VPNEndpoint": {
    "description": "This object contains endpoint abstraction for IP VPN service",
    "type": "object",
    "properties": {
      "EndpointHostname": {
        "description": "This field contains hostname of the router, which terminates VPN for the customer",
        "type": "string"
      },
      "VRF": {
        "description": "This field contains hostname of the router, which terminates VPN for the customer",
        "type": "object",
        "properties": {
          "VRFName": {
            "description": "This field contains VRF name",
            "type": "string"
          },
          "RouteDistinguisher": {
            "description": "This field contains route distinguisher associated with VRF",
            "type": "string",
            "pattern": "^[0-9.:]+$"
          },
          "RouteTargetImport": {
            "description": "This field contains import route target associated with VRF",
            "type": "string",
            "pattern": "^[0-9.:]+$"
          },
          "RouteTargetExport": {
            "description": "This field contains export route target associated with VRF",
            "type": "string",
            "pattern": "^[0-9.:]+$"
          }
        },
        "required": [ "VRFName", "RouteDistinguisher", "RouteTargetImport", "RouteTargetExport" ],
        "additionalProperties": false
      }
```

```
      },
      "required": [ "EndpointHostname", "VRF" ],
      "additionalProperties": false
    }
  }
}
```

The updated JSON schema is now more than twice as big as it was in the preceding section.

For **"properties"**, there is a new key, **"VPNSites"**, which is an array of items. However, items aren't described inline but are instead referenced using the schema keyword **"$ref"**. The schema keyword **definitions** is used to store any kind of data that can be reused anywhere in the core part of the JSON schema. This is very practical, especially in complex JSON schemas, where the same objects or properties might be used many times in different places. (The YANG language has a similar concept called *groupings*, as you will learn in Chapter 13.)

Now let's look at the definitions themselves. As you can see, the structure is precisely the same as in the main JSON schema. **"type": "object"** means that this definition is used for the description of a composite JSON object. This object has its own validation keywords, **"properties"** and **"required"**, precisely following the same logic as the master schema. This example uses *nesting*, which means there might be different levels of the parental relationship between different objects.

According to the logic of Example 11-8, the JSON object **"VPNEndpoint"** has further nesting in the form of the object **"VRF"**, which contains relevant BGP-related information on the IP VPN, including the route distinguisher and import/export route targets. If you are familiar with MPLS/BGP services, you know that route distinguishers and route targets have a specific format and limited range of characters: decimal digits, dots, and colons. That's why you see the additional validation keyword **"pattern"**, which contains the corresponding regex (regular expressions).The regular expressions allow you to perform flexible searches. In this example, the regex is designed to match any text string consisting of numbers and the dot (**.**) and colon (**:**) characters. Therefore, it would match the strings 65000:1 or 10.1.1.1:100, which are examples of IP VPN attributes in this example.

The last new relevant validation keyword is **"additionalProperties"**, which instructs the schema that no additional parameters besides those defined explicitly in the JSON schema are allowed. If any additional parameters are presented in the JSON message, the validation against the schema will fail.

Now that you know the details related to the updates in the JSON schema are explained, we can combine everything discussed to this point. A **"VPNEndpoint"** object is created in definitions, following the standard rules of the JSON schema, and then it is referenced in the main part of the JSON schema as an item within the array. The rationale behind this is that typically there are many endpoints in the IP VPN service, and they are put in an array.

The primary goal of the JSON schema in this example is to validate that the JSON object has the proper format. Example 11-8 shows the JSON object, which is constructed using the provided schema.

Example 11-8 *JSON Object for the JSON Schema of the IP VPN Service*

```
{
  "ContractId": 1,
  "CustomerName": "ACME Corp",
  "CustomerContact": "admin@acme.com",
  "Tags": [ "Very Important Client" ],
  "VPNSites": [
    {
      "EndpointHostname": "us-la-pe-01",
      "VRF": {
        "VRFName": "ACME_Corp",
        "RouteDistinguisher": "65000:1",
        "RouteTargetImport": "65000:101",
        "RouteTargetExport": "65000:101"
      }
    },
    {
      "EndpointHostname": "de-fra-pe-01",
      "VRF": {
        "VRFName": "ACME_Corp",
        "RouteDistinguisher": "65000:1",
        "RouteTargetImport": "65000:101",
        "RouteTargetExport": "65000:101"
```

```
      }
    }
  ]
}
```

Referencing External JSON Schemas

The JSON object in this example has all the possible entries, even optional tags. The **"VPNSites"** array has two objects, each describing one of the VPN termination routers. The JSON schema for customer IP VPN service is almost complete, with only one major part missing: the user-network interface, where the customer is connected to the provider network. It was intentionally not created in the previous section because the interface configuration does not exist only within the IP VPN configuration context; rather, it's independent of anything else. It is therefore possible to create a separate JSON schema for the network interface and then reference it as an external schema in another JSON schema. Example 11-9 provides an overview of a possible JSON schema for interface configuration.

Example 11-9 *Separate JSON Schema for Interface Configuration*

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://network.vendor/interface.schema.json",
  "title": "Interface",
  "description": "This is an arbitrary example of vendor-agnostic router's interface configuration",
  "type": "object",
  "properties": {
    "InterfaceName": {
      "description": "This field contains name of the interface",
      "type": "string"
    },
    "InterfaceMTU": {
      "description": "This field contains MTU of the interface",
      "type": "integer",
          "minimum": 64,
          "maximum": 9216
    },
    "InterfaceDescription": {
      "description": " This field contains description of the interface ",
      "type": "string"
    },
    "InterfaceQoS": {
      "description": "This object contains ingress and egress QoS policies",
      "type": "object",
      "properties": {
        "IngressQoS": {
          "description": " This field contains name of ingress QoS policy ",
          "type": "string"
        },
        "EgressQoS": {
          "description": " This field contains name of egress QoS policy ",
          "type": "string"
        }
      },
      "additionalProperties": false
    },
    "InterfaceEncapsulation": {
```

```
    "description": "This object contains encapsulation relevant parameters",
    "type": "object",
    "oneOf": [
      { "$ref": "#/definitions/EncapsulationNull" },
      { "$ref": "#/definitions/EncapsulationDot1Q" },
      { "$ref": "#/definitions/EncapsulationQinQ" }
    ]
  }
},
"required": [ "InterfaceName", "InterfaceEncapsulation", "InterfaceMTU" ],
"additionalProperties": false,
"definitions": {
  "EncapsulationNull": {
    "description": "This object stands for null encapsulation",
    "properties": {
      "EncapsulationType": {
        "enum": [ "null" ]
      }
    },
    "required": [ "EncapsulationType" ],
    "additionalProperties": false
  },
  "EncapsulationDot1Q": {
    "description": "This object stands for dot1Q encapsulation",
    "properties": {
      "EncapsulationType": {
        "enum": [ "dot1q" ]
      },
      "Vlan1": {
        "type": "integer",
        "minimum": 1,
        "maximum": 4096
      }
    },
    "required": [ "EncapsulationType", "Vlan1" ],
    "additionalProperties": false
  },
  "EncapsulationQinQ": {
    "description": "This object stands for dot1Q encapsulation",
    "properties": {
      "EncapsulationType": {
        "enum": [ "qinq" ]
      },
      "Vlan1": {
        "type": "integer",
        "minimum": 1,
        "maximum": 4096
```

```
        },
        "Vlan2": {
          "type": "integer",
          "minimum": 1,
          "maximum": 4096
        }
      },
      "required": [ "EncapsulationType", "Vlan1", "Vlan2" ],
      "additionalProperties": false
    }
  }
}
```

This new JSON schema looks quite large, but you are already familiar with most of the keywords and concepts (such as nesting). Therefore, in this section we discuss only the new keywords you haven't already seen.

In Example 11-9, the validation keyword **"oneOf"** validates related objects against one of the entries in its array. If one of the objects matches, then validation is successful. As you can see, all the **"one of"** JSON objects in the example have another new validation keyword, **"enum"**, which has an associated value that is an array type. **"enum"** creates in the JSON object a predefined list of values, and only those predefined values can be used. If any other value is used, the validation will fail. Objects don't have to be similar to each other; only one key that has **enum** inside should be the same for all of them. The schema in Example 11-9 shows how convenient this is. **"enum"** is associated with the type of encapsulation, and it is necessary to provide one VLAN (Vlan1) or two VLAN s (Vlan1 and Vlan2) tags for IEEE 802.1q or 802.1ad encapsulations, respectively; no additional information needs to be provided where the encapsulation type is null. This is why different JSON objects in definitions have different entries in the **"required"** validation keyword.

In addition to **"oneOf"** and **"enum"**, there are two more new validation keywords used in conjunction with the integer data type: **"minimum"** and **"maximum"**. **"minimum"** performs a greater-than-or-equal-to operation, whereas **"maximum"** performs a less-than-or-equal-to operation.

Example 11-10 shows three different JSON objects, which can be successfully validated against the provided JSON schema.

Example 11-10 *JSON Objects for the JSON Schema of the Interface*

```
{
  "InterfaceName": "GigabitEthernet0/2/0/10",
  "InterfaceDescription": "Customer ACME Corp",
  "InterfaceEncapsulation": {
    "EncapsulationType": "qinq",
    "Vlan1": 15,
    "Vlan2": 10
  },
  "InterfaceQoS": {
    "IngressQoS": "CUST_in",
    "EgressQoS": "CUST_out"
  },
  "InterfaceMTU": 9000
}



{
  "InterfaceName": "GigabitEthernet0/7/0/11",
  "InterfaceEncapsulation": {
    "EncapsulationType": "dot1q",
    "Vlan1": 1234
  },
  "InterfaceQoS": {
    "IngressQoS": "CUST_in",
```

```
    "EgressQoS": "CUST_out"
  },
  "InterfaceMTU": 9000
}
```

```
{
  "InterfaceName": "GigabitEthernet0/3/0/5",
  "InterfaceEncapsulation": {
    "EncapsulationType": "null"
  },
  "InterfaceMTU": 1514
}
```

As you see, there are some differences between the templates, based on the fact that non-mandatory fields aren't always included. Also, these examples illustrate how an enumeration works.

After the JSON schema for an interface is prepared, it can be referenced in the initial JSON schema for the IP VPN service. Example 11-11 provides a snippet of the changes needed in the initial JSON schema.

Example 11-11 *JSON Objects for the JSON Schema of the Interface with external reference*

```
! The output is truncated for brevity


            "RouteTargetExport": {
              "description": "This field contains export route target associated with VRF",
              "type": "string",
              "pattern": "^[0-9.:]+$"
            }
          },
          "required": [ "VRFName", "RouteDistinguisher", "RouteTargetImport", "RouteTargetExport" ],
          "additionalProperties": false
        },
        "Interfaces": {
          "description": "This object contains configuration of customer interfaces",
          "type": "array",
          "minItems": 1,
          "uniqueItems": true,
          "items": {
            "$ref": "http://network.vendor/interface.schema.json"
          }
        }
      },
      "required": [ "EndpointHostname", "VRF", "Interfaces" ],
      "additionalProperties": false
    }
  }
}
```

In this example you can see a new key **"Interfaces"** in the schema for the IP VPN service, and it has the type array. This array consists of objects that are defined by an external JSON schema referenced in the schema keyword **"$ref"**. The value there is equal to the value of the keyword **"$id"** from Example 11-9. There are two more new validation keywords: **"minItems"** tells the schema there should be at least one element in the array present, and **"uniqueItems"** says that the elements must be unique.

Now we have all the pieces of the puzzle for the JSON schema describing the customer VPN. Example 11-12 shows the JSON object that is

built using these schemas.

Example 11-12 *JSON Object for the Customer IP VPN Schema, Including an External Reference*

```
{
  "ContractId": 1,
  "CustomerName": "ACME Corp",
  "CustomerContact": "admin@acme.com",
  "Tags": [ "Very Important Client" ],
  "VPNSites": [
    {
      "EndpointHostname": "us-la-pe-01",
      "VRF": {
        "VRFName": "ACME_Corp",
        "RouteDistinguisher": "65000:1",
        "RouteTargetImport": "65000:101",
        "RouteTargetExport": "65000:101"
      },
      "Interfaces": [
        {
          "InterfaceName": "GigabitEthernet0/2/0/10",
          "InterfaceDescription": "Customer ACME // HQ",
          "InterfaceEncapsulation": {
            "EncapsulationType": "qinq",
            "Vlan1": 1,
            "Vlan2": 10
          },
          "InterfaceQoS": {
            "IngressQoS": "1Gbps_GOLD_in",
            "EgressQoS": "1Gbps_GOLD_out"
          },
          "InterfaceMTU": 9000
        }
      ]
    },
    {
      "EndpointHostname": "de-fra-pe-01",
      "VRF": {
        "VRFName": "ACME_Corp",
        "RouteDistinguisher": "65000:1",
        "RouteTargetImport": "65000:101",
        "RouteTargetExport": "65000:101"
      },
      "Interfaces": [
        {
          "InterfaceName": "GigabitEthernet0/3/0/13",
          "InterfaceDescription": "Customer ACME // BR1",
          "InterfaceEncapsulation": {
            "EncapsulationType": "dot1q",
            "Vlan1": 123
```

```
        },
        "InterfaceQoS": {
          "IngressQoS": "500Mbps_GOLD_in",
          "EgressQoS": "500Mbps_GOLD_out"
        },
        "InterfaceMTU": 9000
      },
      {
        "InterfaceName": "GigabitEthernet0/5/0/2",
        "InterfaceDescription": "Customer ACME // BR2",
        "InterfaceEncapsulation": {
          "EncapsulationType": "null"
        },
        "InterfaceQoS": {
          "IngressQoS": "500Mbps_GOLD_in",
          "EgressQoS": "500Mbps_GOLD_out"
        },
        "InterfaceMTU": 9000
      }
    ]
  }
 ]
}
```

If the underlying IP/MPLS and BGP network is appropriately configured, the information in Example 11-12 is enough to configure the IP VPN for the customer. Later in this chapter, you will learn how such JSON schemas are applied.

Using JSON Schemas for Data Validation

JSON schemas are used in two major cases. The first case is data modeling. The second case arises when some messages in JSON format are transmitted/received between applications, and applications need to validate JSON objects inside messages to check the following:

- Whether JSON objects have an appropriate format
- Whether all mandatory key/value pairs are present
- Whether there are any illegitimate entries

Validation is the process of comparing a JSON object to a JSON schema and determining whether it matches or not. Typically, the validation process should be built into application call flow when the receiver gets a JSON object. Nevertheless, during the development of various applications, such as network automation, it's useful to have some tools to validate the schema to avoid mistakes or typos and also to verify the JSON objects that are created. A variety of tools have been created for this task using different programming languages; many of them are available online.

The team that is working on standardization of JSON schemas for the IETF has a website (http://json-schema.org/implementations.html) that lists all the available realizations of the JSON schema validations. Example 11-13 shows how to install such a package for Python.

Example 11-13 *Installing a Python Package for JSON Schema Validation*

```
$ pip install jsonschema
Collecting jsonschema
  Downloading
https://files.pythonhosted.org/packages/77/de/47e35a97b2b05c2fadbec67d44cfcdcd09b8086951b331d82de90d2912da/jsonschema-
2.6.0-py2.py3-none-any.whl
Installing collected packages: jsonschema
Successfully installed jsonschema-2.6.0
```

Even if you install this tool by using the Python package installer, it, like many other applications written in Python, can work independently in the operating system, and you can use it outside of Python scripts. This tool is quite simple and works in such a way that there is no output if everything is okay, and it returns an error if something is wrong. This section provides examples for interface configuration for the JSON schema shown in the Example 11-9. Example 11-14 shows how you prepare for JSON schema validation.

Example 11-14 *Preparing for JSON Schema Validation*

```
$ ls -l
-rw-rw-r--. 1 aaa aaa  303 Oct 28 17:40 acme_interfaces.json
-rw-rw-r--. 1 aaa aaa 2843 Oct 28 18:05 interface.schema.json
```

```
$ cat acme_interfaces.json
{
  "InterfaceName": "GigabitEthernet0/2/0/10",
  "InterfaceDescription": "Customer ACME // HQ",
  "InterfaceEncapsulation": {
    "EncapsulationType": "qinq",
    "Vlan1": 172,
    "Vlan2": 10
  },
  "InterfaceQoS": {
    "IngressQoS": "500Mbps_GOLD_in",
    "EgressQoS": "500Mbps_GOLD_out"
  },
  "InterfaceMTU": 9000
}
```

As you can see in Example 11-14, there are two files in the folder: one with the JSON schema and another that is a JSON object that can be transmitted from one application to another. It doesn't contain any mistakes or incorrect entries, so there is no output after validation is conducted, as shown in Example 11-15.

Example 11-15 *Conducting JSON Schema Validation*

```
$ jsonschema -i acme_interfaces.json interface.schema.json
$
```

Now, so you can see the real value in validating JSON objects against a JSON schema, Example 11-16 shows how an object can be modified to provide value out of range.

Example 11-16 *Providing Value Out of Range in a JSON Object and Its Validation*

```
$ cat acme_interfaces.json
{
  "InterfaceName": "GigabitEthernet0/2/0/10",
  "InterfaceDescription": "Customer ACME // HQ",
  "InterfaceEncapsulation": {
    "EncapsulationType": "qinq",
    "Vlan1": 172,
    "Vlan2": 10000
  },
  "InterfaceQoS": {
    "IngressQoS": "500Mbps_GOLD_in",
    "EgressQoS": "500Mbps_GOLD_out"
  },
  "InterfaceMTU": 9000
}
```

```
$ jsonschema -i acme_interfaces.json interface.schema.json
{'EncapsulationType': 'qinq', 'Vlan1': 172, 'Vlan2': 10000}: {'EncapsulationType': 'qinq', 'Vlan1': 172, 'Vlan2': 10000}
is not valid under any of the given schemas
```

Because VLAN is a 12-bit field, the possible range for VLANs is 1 to 4096. (Hexadecimal values of 0x000 and 0xFFF are reserved.) You saw this constraint imposed using validation keywords in the JSON schema earlier in this chapter. Therefore, when the validation is launched, you see output, which indicates that the provided output doesn't match any schemas.

Example 11-17 shows another case in which there is a typo in one of the keys.

Example 11-17 *A Typo in a Key in a JSON Object and Its Validation*

```
$ cat acme_interfaces.json

{

  "InterfaceName": "GigabitEthernet0/2/0/10",

  "InterfaceDescription": "Customer ACME // HQ",

  "InterfaceEncapsulation": {

    "EncapsulationType": "qinq",

    "Vlan1": 172,

    "Vlan2": 10

  },

  "InterfaceQoS": {

    "IngressQoS": "500Mbps_GOLD_in",

    "EgressQoS": "500Mbps_GOLD_out"

  },

  "IInterfaceMTU": 9000

}




$ jsonschema -i acme_interfaces.json interface.schema.json

{'InterfaceName': 'GigabitEthernet0/2/0/10', 'InterfaceDescription': 'Customer ACME // HQ', 'InterfaceEncapsulation':
{'EncapsulationType': 'qinq', 'Vlan1': 172, 'Vlan2': 10}, 'InterfaceQoS': {'IngressQoS': '500Mbps_GOLD_in', 'EgressQoS':
'500Mbps_GOLD_out'}, 'IInterfaceMTU': 9000}: 'InterfaceMTU' is a required property

{'InterfaceName': 'GigabitEthernet0/2/0/10', 'InterfaceDescription': 'Customer ACME // HQ', 'InterfaceEncapsulation':
{'EncapsulationType': 'qinq', 'Vlan1': 172, 'Vlan2': 10}, 'InterfaceQoS': {'IngressQoS': '500Mbps_GOLD_in', 'EgressQoS':
'500Mbps_GOLD_out'}, 'IInterfaceMTU': 9000}: Additional properties are not allowed ('IInterfaceMTU' was unexpected)
```

The output in Example 11-17 indicates that there are two errors. First, it shows that the required key **"InterfaceMTU"** is missing. Next, it shows that there are no additional parameters allowed, which means the typo in the key is treated as an additional parameter. You can see the same error in Example 11-18, which shows new key added intentionally.

Example 11-18 *An Unexpected Key in a JSON Object and Its Validation*

```
$ cat acme_interfaces.json

{

  "InterfaceName": "GigabitEthernet0/2/0/10",

  "InterfaceDescription": "Customer ACME // HQ",

  "InterfaceEncapsulation": {

    "EncapsulationType": "qinq",

    "Vlan1": 172,

    "Vlan2": 10

  },

  "InterfaceQoS": {

    "IngressQoS": "500Mbps_GOLD_in",

    "EgressQoS": "500Mbps_GOLD_out"

  },

  "InterfaceMTU": 9000,

  "SomeStrangeKey": "SomeStrangeValue"

}
```

```
$ jsonschema -i acme_interfaces.json interface.schema.json

{'InterfaceName': 'GigabitEthernet0/2/0/10', 'InterfaceDescription': 'Customer ACME // HQ', 'InterfaceEncapsulation':
{'EncapsulationType': 'qinq', 'Vlan1': 172, 'Vlan2': 10}, 'InterfaceQoS': {'IngressQoS': '500Mbps_GOLD_in', 'EgressQoS':
'500Mbps_GOLD_out'}, 'InterfaceMTU': 9000, 'SomeStrangeKey': 'SomeStrangeValue'}: Additional properties are not allowed
('SomeStrangeKey' was unexpected)
```

Such behavior isn't the default in a JSON schema. By default, such additional key/pair values are ignored and conveyed to the application. To prevent this behavior, the JSON schema defined in Example 11-9 has the validation keyword **"additionalProperties"** set to **false**.

Example 11-19 shows an error that is caused by using a value that is out of the defined range.

Example 11-19 *Using a Value That Is Out of Range in a JSON Object and Its Validation*

```
$ cat acme_interfaces.json

{

  "InterfaceName": "GigabitEthernet0/2/0/10",

  "InterfaceDescription": "Customer ACME // HQ",

  "InterfaceEncapsulation": {

    "EncapsulationType": "qinq",

    "Vlan1": 172,

    "Vlan2": 10

  },

  "InterfaceQoS": {

    "IngressQoS": "500Mbps_GOLD_in",

    "EgressQoS": "500Mbps_GOLD_out"

  },

  "InterfaceMTU": 90000

}
```

```
$ jsonschema -i acme_interfaces.json interface.schema.json

90000: 90000 is greater than the maximum of 9216
```

The error message in Example 11-19 is different from the one in Example 11-16 because in Example 11-16, it is not a single value (**Vlan2**) that is compared to its range but a bundle of parameters (**EncpasulationType, Vlan1**, and Vlan2). If there is theoretically another **enum** entry, which has another predefined range for VLANs, it will work. In Example 11-19, it's clearly stated which value is out of range and in which direction (higher than the maximum).

Performing validation ensures that an application receives a consistent set of information in a proper format. Validation of JSON objects against JSON schemas can in some cases protect your application from malicious key/value pairs and increase the security of your objects.

Summary

In this chapter, you have learned about the JSON data format and JSON schemas, how they are built, and where they are used. This chapter covers the following details:

- The simple, powerful, language-independent JSON data format
- JSON's six data types (four primary data types and two nested data types)
- Support for JSON format in current programming and scripting languages
- The importance of paying attention to quotation marks when working with JSON data
- JSON schemas, which are used to provide a structured context for JSON objects
- JSON schemas, which define which key/pair values and in what format should be presented in a JSON object
- Reusing defined data structures called definitions and referencing external JSON schemas
- Nesting objects in a hierarchy
- Validation of JSON objects against JSON schemas to ensure that a receiving application obtains a consistent set of information in proper format

Chapter 12 discusses YAML.

Chapter 12. YAML

YAML Ain't Markup Language (*YAML*) is a user-friendly data serialization language that is useful for engineers working with data and building device configurations. YAML was first proposed in 2001 by Clark Evans. As per the initial draft, YAML was said to stand for "Yet Another Markup Language," but the name was later revised to "YAML Ain't Markup Language." As the name suggests, the creators of YAML didn't want it to become yet another random markup language but wanted to put more emphasis on data representation and human readability of data. YAML uses Unicode printable characters and has minimal structural characters, which means it is easy to use YAML to represent and understand data in a meaningful way. To date, three versions of YAML have been released, the latest of which is Version 1.2. The latest specification of YAML can be found at www.yaml.org. YAML is well supported by modern programming languages such as Python and Java.

Why do we really need another data representation language when there are already well-known data representation formats such as Extensible Markup Language (XML) and JavaScript Object Notation (JSON), which are heavily used in most network and web-based applications? After reading through this chapter and working with YAML you will realize that it is inherently easier to read and write data with YAML. Because of its exceptionally human-readable way of representing data, YAML has become an important language for IT operations and automation tools such as Ansible. Before digging into the fundamentals of YAML, let's examine the key differences between XML, JSON, and YAML:

• **Data representation**: XML is a markup language, whereas JSON and YAML are data formats. XML uses tags to define the elements and stores data in a tree structure, whereas data in JSON is stored like a map with key/value pairs. YAML, on the other hand, allows representation of data both in list or sequence format and in the form of a map with key/value pairs. JSON and YAML uses different indentation styles: JSON uses tabs, whereas YAML uses a hyphen (-) followed by whitespace.

• **Comments**: Comments makes it easier to understand and interpret data. Whereas JSON has no concept of comments, XML allows you to add comments within a document. YAML was designed for readability and thus allows comments.

• **Data types**: XML supports complex data types such as charts, images, and other non-primitive data types. JSON supports only strings, numbers, arrays, Booleans, and objects. YAML, on the other hand, supports complex data types such as dates and time stamps, sequences, nested and recursive values, and primitive data types.

• **Data readability**: It is difficult to read and interpret data written in XML, but it is fairly easy to interpret data in JSON format, and it is much easier to read data in YAML than in JSON format.

• **Usability and purpose**: XML is used for data interchange (that is, when a user wants to exchange data between two applications). JSON is better as a serialization format and is used for serving data to application programming interfaces (APIs). YAML is best suited for configuration.

• **Speed**: XML is bulky and slow in parsing, leading to relatively slow data transmission. JSON files are considerably smaller than XML files, and JSON data is quickly parsed by the JavaScript engine, enabling faster data transmission. YAML, as a superset of JSON, also delivers faster data transmission, but it's important to remember that JSON and YAML are used in different scenarios.

YAML data is stored in a file that has .yaml or .yml file extension. YAML includes important constructs that distinguish the language from the document markup, and thus there are two basic rules to remember when creating a YAML file:

• YAML is case sensitive.

• YAML does not allow the use of tabs. Use spaces instead.

As mentioned earlier, YAML provides support for various programming languages. In this chapter, we focus on using YAML-based files with Python. Later in this chapter, you will see how to use YAML files in Python code. Before we get there, the next section focuses on how different types of data are stored in YAML files.

YAML Structure

YAML allows you to easily and quickly format data in human-readable format. A YAML file begins with three hyphens (---), which separate the directives from the document. A single YAML file can contain multiple YAML documents, each beginning with three hyphens. Similarly, three dots (...) represent the end of a document (but do not start a new one). You can start a new YAML document without closing the previous one. Example 12-1 illustrates the beginning and closing of a document. In the first section of the example, notice that multiple documents are started within the same file, without the previous documents being closed. In the second section of the example, each document is closed before another document is started. Example 12-1 also illustrates the use of comments, which start with the pound sign (#). You can add comments anywhere in a document: at the beginning of the document, between the directives, or even on the same line as directives.

Example 12-1 *Starting and Closing Documents in YAML*

```
# Routing Protocols on a device

---

- OSPF

- EIGRP

- BGP


# Interfaces participating in OSPF

---
```

```
- Ethernet1/1     # Connected to Gi0/1 on R11

- Ethernet1/2     # Connected to Gi0/2 on R12

- Ethernet1/3     # Connected to Gi0/3 on R13


# Interfaces participating in EIGRP

---

- Ethernet1/4

- Ethernet1/5
```

```
# Network Events

---

- datetime: 05/17/20202 20:03:00

- event: OSPF Flap

- intf: Ethernet1/1

...

---

- time: 05/18/20202 13:45:07

- event: OSPF Flap

- intf: Ethernet1/3

...
```

YAML represents native data structures using a simple identifier called a *tag*; a single native data structure is called a *node*. Tagged nodes are rooted and connected using directed graphs, and all nodes can be reached from the root node. A node can have content in one of the three formats:

• **Sequence**: An ordered series of zero or mode nodes that may contain the same node more than once or that may even contain itself.

• **Mapping**: An ordered set of key/value node pairs.

• **Scalar**: An opaque datum that can be presented as a series of zero or more Unicode characters.

YAML Sequences and mappings are both categorized as collections. Let's now look at how data can be represented in YAML using these different formats.

Collections

YAML's block collections include sequences and mappings. When you define a block collection, you begin each entry on its own line and use indentation to define the scope of the block. You define a block sequence by beginning an entry with a hyphen (-) followed by whitespace. A mapping is a simple representation of data in key/value pairs, where a key is followed by a colon (:), a space, and then the value. Example 12-2 shows sequences and mappings in YAML.

Example 12-2 *Sequences and Mappings in YAML*

```
# Sequence of Scalars

# This sequence contains the names of nodes in the network

- rtr-R1-ios

- rtr-R2-ios-xr

- rtr-R3-nx-os
```

```
# Mapping scalars to scalars

# Below mapping represents node information such as name, OS and software version

name: rtr-R1-ios

OS: IOS-XE

version: 17.1
```

Note

Maps don't have to have string keys. A key in a map can be of type integer, float, or another type. The following is an example of a map with a float key:

```
0.11: a float key
```

Refer to yaml.org for more details.

Keys in a map can also be complex. You can use a question mark (?) followed by a space to represent a complex key; the key can span multiple lines. Example 12-3 illustrates how to create complex key for mapping.

Example 12-3 *Complex Key for Mapping in YAML*

```
# Complex Key

? |

  This is a Complex key

  and span across multiple lines

: this is its value
```

YAML also provides the flexibility to represent one format of data in another format—that is, to perform cross-formatting. For instance, you can define a mapping of sequences or a sequence of mappings, as shown in Example 12-4.

Example 12-4 *Mapping of Sequences and Sequence of Mappings in YAML*

```
# Mapping of Sequences

Nodes:

  - rtr-R1

  - rtr-R2

Intf:

  - GigabitEthernet0/1

  - GigabitEthernet0/2

  - GigabitEthernet0/3
```

```
# Sequence of Mappings

-

  name: rtr-R1

  mgmt-ip: 192.168.1.1

  user: admin

-

  name: rtr-R2

  mgmt-ip: 192.168.1.2

  user: cisco
```

Along with cross-formatting, YAML supports the flow style of data representation. In the flow style, YAML uses explicit indicators instead of using indentation to denote scope. You write a flow sequence inside square brackets as comma-separated values, and you write a flow mapping inside curly braces. Example 12-5 illustrates the representation of data in a flow sequence and in a flow mapping. A flow sequence can also be understood as a sequence of sequences, and a flow mapping can be understood as a mapping of mappings.

Example 12-5 *FlowSequence and FlowMapping in YAML*

```
# Flow Sequence

- [node-name, mgmt-ip, user  ]

- [rtr-R1, 192.168.1.1, admin]

- [rtr-R2, 192.168.1.2, cisco]
```

```
# Flow Mapping

rtr-R1: {mgmt-ip: 192.168.1.1, user: admin}

rtr-R2: {

    mgmt-ip: 192.168.1.2,
```

```
    user: cisco

  }
```

YAML also allows mapping between sequences using complex keys. However, some parsers may return syntax errors with such mappings. Example 12-6 demonstrates the creation of a mapping between sequences using complex keys. This example shows a sequence of node names that are mapped to management IP addresses.

Example 12-6 *Mapping Between Sequences Using Complex Keys in YAML*

```
# Mapping Between Sequences using Complex Keys

? - rtr-R1

  - rtr-R2

: [192.168.1.1, 192.168.1.2]
```

Because YAML is a superset of JSON, you can create JSON-style maps and sequences in YAML. Example 12-7 shows a JSON-style representation of a sequence and maps in YAML.

Example 12-7 *JSON-Style Sequence and Maps in YAML*

```
# JSON Style Sequence and Maps

json_map: {"key": "value"}

json_seq: [3, 2, 1, "get set go"]

quotes optional: {key: [3, 2, 1, get set go]}
```

Scalars

YAML scalars are either written in literal block or in folded block styles. YAML scalars written in literal block format use the pipe character (**|**) literal, whereas scalars written in folded block style use the greater-than symbol (>). Literal block style preserves line breaks, and the literal continues until the end of the indentation and the leading indentation is stripped. If there are more indented lines, the following lines use the same indentation rule. Similarly, in a folded block, each line break is folded to a space unless it ends an empty or a more-indented line. For instance, a multiple-line string can be either written as a literal block using | or as a folded block using >, as shown in Example 12-8.

Example 12-8 *Scalars with Literal Block Style and Folded Block Style in YAML*

```
---

literal_block: |

    This entire block of text is the value of 'literal_block' key,

    with line breaks being preserved.


    You can keep adding more lines of text below and maintain the same

    Indentation as above.


         The 'More indented' lines are represented in the following manner -

         In this case, lines following the above line have same indentation.
```

```
---

folded_style: >

 This entire block of text is the value of 'folded_style' key.

 each linebreak is represented by a space.
```

Note

It is important to remember that indentation defines the scope of a block.

YAML also include flow scalars, which are represented in plain style or quoted style. The double-quoted style provides escape sequencing, whereas single-quoted style is applicable when escaping is not needed. All flow scalars can span multiple lines. Note that the line breaks are always folded. Example 12-9 illustrates flow scalars using different styles.

Example 12-9 *FlowScalars in YAML*

```
# Plain style flow scalar

plain:
```

```
 This is an unquoted scalar and

 it spans across multiple lines.


# Single quoted scalar

random: 'This is a single quoted text'

str: 'This is not''a # comment''.'


# Double quoted flow scalar

double-quoted: "this is a

 Quoted scalar.\n"

unicode: "The code looks good.\u263A"
```

Tags

YAML tags are used to represent native data structures as well as other data formats, such as sets and ordered maps. In addition, YAML has root nodes as well as nested nodes. The root object or root node, which continues for an entire document, is represented as a map that is equivalent to a dictionary or an object in a different language. When representing numeric data, you generally make use of integer or floating point data types, and such data is represented in the form of integer or floating point tags. Example 12-10 illustrates the representation of integer and floating point tags. Note that the integer tags include integer values, and floating point tags include decimal and exponential values.

Example 12-10 *Integer and Floating Point Tags*

```
# Integer Tags

canonical: 12345

decimal: +12789

octal: 0o17

hexadecimal: 0xEF
```

```
# Floating Point Tags

canonical: 1.23015e+3

exponential: 12.6415e+02

fixed: 9870.26

negative infinity: -.inf

not a number: .NaN
```

You can represent null, string, and Boolean values in YAML. These fall under the category of miscellaneous tags. Example 12-11 illustrates the representation of such types of data. There are a few points to remember when dealing with Boolean and string data types. A Boolean value cannot be represented as 1 as 1 is interpreted as a number, not a Boolean value. Similarly, string data is not required to be quoted but can be. Note that the null key or value can be used to represent null data being returned by the data source.

Example 12-11 *Miscellaneous Tags*

```
# Miscellaneous Tags

null:

null_value: null

true: y

false: n

booleans: [ true, false ]

string: 'hello world'
```

YAML can also parse some extra types, such as ISO-formatted date and datetime types. Usually it is possible to parse date or datetime only in programming languages or databases, but YAML allows you to represent date and time-related data by using date and datetime literals. Example 12-12 demonstrates how to use date and datetime literals in YAML. In Example 12-12, notice that date and datetime data are represented in different ways.

Example 12-12 *Date and Datetime YAML Types*

```
---
```

```
datetime: 2020-2-15T05:45:23.1Z

datetime_with_spaces: 2019-10-17 00:02:43.10 -5

date: 2020-05-13
```

In YAML, there are also explicit tags and global tags. Explicit tags are denoted using the exclamation point (!) symbol. Explicit tags are used to explicitly represent particular data formats. For instance, the **!binary** explicit tag indicates that a string is actually a Base64-encoded representation of a binary blob. Global tags, on the other hand, are URIs and may be represented using a tag shorthand notation. Example 12-13 illustrates the representation of both explicit and global tags. Note that the primary tag handle is an exclamation point (!) symbol. Thus, the explicit tag for a string is represented in YAML as **!!str**.

Example 12-13 *Explicit and Global Tags in YAML*

```
# Explicit Tags

date-string: !!str 2020-05-10


gif-file: !!binary |

  R0lGODlhDAAMAIQAAP//9/X17unp5WZmZgAAAOfn515eXvPz7Y6OjuDg4J+fn5

  OTk6enp56enmlpaWNjY6Ojo4SEhP/++f/++f/++f/++f/++f/++f/++f/++f/+

  +f/++f/++f/++f/++f/++SH+Dk1hZGUgd2l0aCBHSU1QACwAAAAADAAMAAAFLC

  AgjoEwnuNAFOhpEMTRiggcz4BNJHrv/zCFcLiwMWYNG84BwwEeECcgggoBADs=
```

```
# Global Tags


%TAG ! tag:network.local,8080:

--- !nodes

  # Use the ! handle for presenting

  # tag:network.local,8080:node1

- !node1

 mgmt-ip: 192.168.1.1

 username: admin

 password: cisco

- !node1

 mgmt-ip: 192.168.1.2

 username: admin

 password: cisco!123
```

YAML also supports unordered sets and ordered mappings. A set is represented as a mapping where each key is associated with a null value. Ordered maps are represented as a sequence of mappings, with each mapping having one key. Example 12-14 demonstrates how to define unordered sets and ordered mappings in a YAML file.

Example 12-14 *Unordered Sets and Ordered Mappings in YAML*

```
# Sets

--- !!set

? item1

? item2

? item3


# Sets can also be written without any explicit tag

set1:

  ? item1

  ? item2

  ? item3
```

```
or: {item1, item2, item3}


# Sets are just maps with null values. set1 can also be represented as below

set2:

  item1: null

  item2: null

  item3: null
```

```
# Ordered Maps

# Below map lists the items and its prices

--- !!omap

- item1: 67

- item2: 64

- item3: 51
```

Anchors

Working with repetitive content can be tedious and tiresome, especially when you're dealing with larger data sets and when certain section of data are being repeated multiple times. To overcome such challenges, YAML provides a very handy feature known as *anchors*. An anchor acts as a reference pointer to data already defined once in the document; that reference pointer can be used multiple times later in the YAML configuration files to duplicate the same data. For instance, you can create an anchor for a value of a key and reference that anchor at another place in the document for another key to ensure that the second key has the same values as the first key. An anchor is defined by using an ampersand (&) followed by the anchor name. After the anchor name is defined, it can be referenced at another place by using *aliases*, which are represented by an asterisk (*****) followed by the specified anchor name. Example 2-15 illustrates the use of anchors to duplicate the content in a YAML file.

Example 12-15 *Using Anchors in YAML*

```
# Using Anchors

key1: &val1 This is the value of both key1 and key2


key2: *val1
```

Anchors can not only be used for duplicating data but can also be used for duplicating or inheriting properties. This is achieved in YAML by using *merge keys*. A merge key, represented by a double less-than symbol (**<<**), is used to indicate that all the keys of one or more specified maps will be inserted into the current map. Example 12-16 demonstrates the use of merge keys in YAML.

Example 12-16 *Using Merge Key in YAML*

```
# Merge Key

login: &login

  user: admin

  pass: cisco




node1: &node1

  <<: *login

  mgmt-ip: 192.168.1.1


node: &node2

  <<: *login

  mgmt-ip: 192.168.1.2
```

YAML Example

So far, this chapter has described the fundamentals of YAML and how various YAML data representation techniques are used to represent data in a human-readable format. Now it is time to put your new knowledge into practice and create a YAML file that makes it easier to

understand the data. Example 12-17 shows a configuration file for setting up and enabling networking for network devices and virtual cloud servers in a small data center. You could use this configuration with automation tools for one-click deployment of servers in the network. In this example, default credentials are defined for servers as well as network devices such as routers. Then, under the networking node, domain name and domain name server information that is going to be same across the network is specified. Under the networking node are subnodes for routers and host networks, which are then further expanded to various host profiles, such as tenant, management, storage, API, and so on. Each profile represents a specific configuration for that segment of the network. This example also shows a very efficient use of merge keys to duplicate the credentials at different places in the configuration file.

Example 12-17 *YAML Network Configuration Example*

```
SERVER_COMMON: &srvrcred {server_username: root, password: cisco!123}

DEFAULT_CRED: &login

  - user: admin

    password: Cisco@123

  - user: network-admin

    password: Cisco!123


NETWORKING:

  domain_name: cisco.com

  domain_name_servers: [172.16.101.1]

  routers:

  - pool: [192.168.1.2 to 192.168.1.253]

    gateway: 192.168.1.1

    <<: *login

  host_networks:

  - gateway: 192.168.50.1

    pool: [192.168.50.10 to 192.168.50.250]

    segments: [management, provision]

    subnet: 192.168.50.0/24

    vlan_id: 50

    <<: *srvrcred

  - gateway: 10.117.0.1

    pool: [10.117.0.5 to 10.117.0.254]

    segments: [tenant]

    subnet: 10.117.0.0/24

    vlan_id: 117

    <<: *srvrcred

  - gateway: 10.118.0.1

    pool: [10.118.0.5 to 10.118.0.254]

    segments: [storage]

    subnet: 10.118.0.0/24

    vlan_id: 118

  - gateway: 10.35.22.1

    segments: [api]

    subnet: 10.35.22.0/24

    vlan_id: 3522
```

Handling YAML Data Using Python

All data formats are supported by and can be integrated with various programming languages. YAML is a data serialization format designed for better human readability and easy interaction with scripting languages, and it is supported by Python. When dealing with specific format of data, such as XML or JSON, a basic requirement for a library in any programming language is a parser. Python has a YAML parser known as

PyYAML. PyYAML is a complete YAML parser for YAML Version 1.1 that supports YAML tags and provides Python-specific tags that allow programmers to represent arbitrary Python objects.

Before you can work with the PyYAML package, you need to install PyYAML. Example 12-18 demonstrates the installation of the PyYAML package inside a virtual environment. When the package is installed, you can go to the Python terminal and import the package named yaml. You can use the **dir()** method to find all the properties and methods available for the yaml object. Some of the common methods are highlighted in Example 12-18.

Example 12-18 *Installing PyYAML and Exploring the PyYAML Package*

```
[root@node1 ~]# virtualenv yaml
Using base prefix '/usr'
  No LICENSE.txt / LICENSE found in source
New python executable in /root/yaml/bin/python3
Also creating executable in /root/yaml/bin/python
Installing setuptools, pip, wheel...
done.
[root@node1 ~]# source yaml/bin/activate
(yaml) [root@node1 ~]# pip3 install pyyaml
Collecting pyyaml
  Downloading PyYAML-5.3.1.tar.gz (269 kB)


|,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,
ñà,ñà,ñà,ñà,ñã| 266 kB 10.7 MB/s eta 0:00:
|,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,ñà,
ñà,ñà,ñà,ñà,ñà| 269 kB 10.7 MB/s
Building wheels for collected packages: pyyaml
  Building wheel for pyyaml (setup.py) ... done
  Created wheel for pyyaml: filename=PyYAML-5.3.1-cp36-cp36m-linux_x86_64.whl
size=44621
sha256=10808863673407f7436cc15e471caf30cd06fd1462ac9f7f512d3c431ef37509
  Stored in directory:
/root/.cache/pip/wheels/e5/9d/ad/2ee53cf262cba1ffd8afe1487eef788ea3f260b7e6232a80
fc
Successfully built pyyaml
Installing collected packages: pyyaml
Successfully installed pyyaml-5.3.1
(yaml) [root@node1 ~]#
[root@node1 demo]# python3
Python 3.7.6 (default, Dec 30 2019, 19:38:28)
[Clang 11.0.0 (clang-1100.0.33.16)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import yaml
>>> dir(yaml)
['AliasEvent', 'AliasToken', 'AnchorToken', 'BaseDumper', 'BaseLoader',
'BlockEndToken', 'BlockEntryToken', 'BlockMappingStartToken',
'BlockSequenceStartToken', 'CollectionEndEvent', 'CollectionNode',
'CollectionStartEvent', 'DirectiveToken', 'DocumentEndEvent', 'DocumentEndToken',
'DocumentStartEvent', 'DocumentStartToken', 'Dumper', 'Event', 'FlowEntryToken',
```

```
'FlowMappingEndToken', 'FlowMappingStartToken', 'FlowSequenceEndToken',

'FlowSequenceStartToken', 'FullLoader', 'KeyToken', 'Loader', 'MappingEndEvent',

'MappingNode', 'MappingStartEvent', 'Mark', 'MarkedYAMLError', 'Node',

'NodeEvent', 'SafeDumper', 'SafeLoader', 'ScalarEvent', 'ScalarNode',

'ScalarToken', 'SequenceEndEvent', 'SequenceNode', 'SequenceStartEvent',

'StreamEndEvent', 'StreamEndToken', 'StreamStartEvent', 'StreamStartToken',

'TagToken', 'Token', 'UnsafeLoader', 'ValueToken', 'YAMLError',

'YAMLLoadWarning', 'YAMLObject', 'YAMLObjectMetaclass', '__builtins__',

'__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',

'__path__', '__spec__', '__version__', '__with_libyaml__', '_warnings_enabled',

'add_constructor', 'add_implicit_resolver', 'add_multi_constructor',

'add_multi_representer', 'add_path_resolver', 'add_representer', 'compose',

'compose_all', 'composer', 'constructor', 'dump', 'dump_all', 'dumper', 'emit',

'emitter', 'error', 'events', 'full_load', 'full_load_all', 'io', 'load',

'load_all', 'load_warning', 'loader', 'nodes', 'parse', 'parser', 'reader',

'representer', 'resolver', 'safe_dump', 'safe_dump_all', 'safe_load',

'safe_load_all', 'scan', 'scanner', 'serialize', 'serialize_all', 'serializer',

'tokens', 'unsafe_load', 'unsafe_load_all', 'warnings']
```

This section focuses on some of the most common methods from the yaml package that can be used when dealing with YAML data. The first function we look at is the **yaml.dump()** function, which you use to serialize a Python object into a YAML stream. In other words, this function allows you to convert data into a more human-readable format. It produces a YAML document as a UTF-8-encoded **str** object. You can change the encoding by specifying the encoding attribute as a parameter in the **yaml.dump()** function. Example 12-19 demonstrates how to use the **yaml.dump()** method to represent some data about a user in a more readable format. The first section of the example shows some information saved inside a data variable that is in an understandable format; the second section of the example shows that serializing the data into a YAML stream makes the data even easier to understand.

Example 12-19 *Serializing a Python Object into a YAML Stream by Using the* ***yaml.dump()*** *Method*

```
! demo.py

import yaml


data = {

    "Name": "John Doe",

    "Age": 40,

    "Company": "XYZ",

    "Social-media": {'Facebook': "johndoe",

                     "twitter": "johndoe"},

    "Job Title": "Network Automation Engineer",

    "Skills": ["Python", "Nexus", "IOS-XE", "YAML", "JSON"]

}


print(yaml.dump(data))
```

```
[root@node1 demo]# python demo.py

Age: 40

Company: XYZ

Job Title: Network Automation Engineer

Name: John Doe

Skills:

- Python
```

```
- Nexus
- IOS-XE
- YAML
- JSON
Social-media:
  Facebook: johndoe
  twitter: johndoe
```

The **yaml.dump()** method is used to serialize a Python object and also to write the serialized YAML stream into a file. Two parameters are passed in the **yaml.dump()** function: the Python object to be converted in the YAML stream and the file to which the stream will be written (see Example 12-20).

Example 12-20 *Saving a YAML Stream to a File*

```
! demo.py
import yaml


data = {
    "Name": "John Doe",
    "Age": 40,
    "Company": "XYZ",
    "Social-media": {'Facebook': "johndoe",
                     "twitter": "johndoe"},
    "Job Title": "Network Automation Engineer",
    "Skills": ["Python", "Nexus", "IOS-XE", "YAML", "JSON"]
}


with open('test.yaml', 'w') as f:
    yaml.dump(data, f)
```

```
[root@node1 demo]# python3 demo.py
[root@node1 demo]# cat test.yaml
Age: 40
Company: XYZ
Job Title: Network Automation Engineer
Name: John Doe
Skills:
- Python
- Nexus
- IOS-XE
- YAML
- JSON
Social-media:
  Facebook: johndoe
  twitter: johndoe
```

The **yaml.dump()** method also allows you to sort a data stream based on keys. The **sort_keys** attribute, when set to **True**, sorts the keys in the YAML stream; when set to **False**, it presents the data as it was inserted in the YAML stream. Example 12-21 demonstrates the difference between sorted and unsorted YAML data streams.

Example 12-21 *Sorting YAML Data Streams*

```
import yaml


data = {
    "Name": "John Doe",
    "Age": 40,
    "Company": "XYZ",
    "Social-media": {'Facebook': "johndoe",
                     "twitter": "johndoe"},
    "Job Title": "Network Automation Engineer",
    "Skills": ["Python", "Nexus", "IOS-XE", "YAML", "JSON"]
}
print("---------Sorted----------")
print(yaml.dump(data, sort_keys=True))
print("--------Unsorted---------")
print(yaml.dump(data, sort_keys=False))
```

```
[root@node1 demo]# python3 demo.py
---------Sorted----------
Age: 40
Company: XYZ
Job Title: Network Automation Engineer
Name: John Doe
Skills:
- Python
- Nexus
- IOS-XE
- YAML
- JSON
Social-media:
  Facebook: johndoe
  twitter: johndoe


--------Unsorted---------
Name: John Doe
Age: 40
Company: XYZ
Social-media:
  Facebook: johndoe
  twitter: johndoe
Job Title: Network Automation Engineer
Skills:
- Python
- Nexus
- IOS-XE
- YAML
- JSON
```

Another important function in the yaml package is the **yaml.load()** method, which accepts a byte string, a Unicode string, an open binary file object, or an open text file object as the parameter and returns a Python object. In other words, it converts a YAML document into a Python object. Example 12-22 demonstrates how the **yaml.load()** function returns a Python object. In this example, the data.yaml file contains a mapping of sequences of routers and switches. The demo.py file opens the file data.yaml, and the object is passed as a parameter to the **yaml.load()** function. Notice that this example also has the **Loader** attribute set to **yaml.FullLoader**. The **FullLoader** parameter handles the conversion from YAML scalar values to Python dictionary format.

Example 12-22 *Using the **yaml.load()** Function*

```
! data.yaml

routers:

  - rtr-ios-R1

  - rtr-xr-R2

  - rtr-nx-R3

switches:

  - sw-ios-SW1

  - sw-xe-SW2

  - sw-nx-SW3
```

```
! demo.py

import yaml


with open('data.yaml') as f:

    data = yaml.load(f, Loader=yaml.FullLoader)


print(data)

print("\n----------YAML Serialized Format------------\n")

print(yaml.dump(data))
```

```
[root@node1 demo]# python3 demo.py

{'routers': ['rtr-ios-R1', 'rtr-xr-R2', 'rtr-nx-R3'], 'switches': ['sw-ios-SW1',

'sw-xe-SW2', 'sw-nx-SW3']}


----------YAML Serialized Format------------


routers:

- rtr-ios-R1

- rtr-xr-R2

- rtr-nx-R3

switches:

- sw-ios-SW1

- sw-xe-SW2

- sw-nx-SW3
```

Note

For demonstration purposes, some examples in this section use the **yaml.load()** function. However, it is not recommended to use the **yaml.load()** function when data is received from an external and untrusted source. It is thus recommended to use the **yaml.safe_load()** function. Alternatively, you can use the **yaml.load()** function but with the **Loader** value set to **yaml.SafeLoader**.

The **yaml.load()** function is useful when you need to parse through a single YAML document. However, if a string or a YAML file contains multiple documents, you should use the **yaml.load_all()** function to load all the documents. Example 12-23 illustrates the use of the **yaml.load_all()** function to load all the documents in a YAML file.

Example 12-23 *Loading Multiple YAML Documents by Using the* ***yaml.load_all()*** *Function*

```
! data.yaml

# First YAML Document

network-devices:

  routers:

  - rtr-ios-R1

  - rtr-xr-R2

  - rtr-nx-R3

  switches:

  - sw-ios-SW1

  - sw-xe-SW2

  - sw-nx-SW3

---

# Second YAML Document

hosts:

  - srvr1: "linux"

    patched: True

    OS: CentOS

  - srvr2: "Microsoft Windows"

    patched: False

    OS: "Windows 10"
```

```
! demo.py

import yaml


with open('data.yaml') as f:

    # yaml.load_all() loads all the YAML documents present in a YAML file

    # and returns an object, which can be further iterated

    docs = yaml.load_all(f, Loader=yaml.FullLoader)

    for doc in docs:

        for i,j in doc.items():

            print(i, "->", j)
```

```
[root@node1 demo]# python3 demo.py

network-devices -> {'routers': ['rtr-ios-R1', 'rtr-xr-R2', 'rtr-nx-R3'],

'switches': ['sw-ios-SW1', 'sw-xe-SW2', 'sw-nx-SW3']}

hosts -> [{'srvr1': 'linux', 'patched': True, 'OS': 'CentOS'}, {'srvr2':

'Microsoft Windows', 'patched': False, 'OS': 'Windows 10'}]
```

The YAML package also provides access to lower-level APIs when parsing YAML files. The **yaml.scan()** method scans through a YAML file and generates tokens that can be used to understand the kind of YAML data being worked on. As described earlier in this chapter, data is presented in YAML in three forms:

• Scalar

• Sequence

• Mapping

The tokens generated by the **scan()** method list the start and end tokens when parsing any of these three forms of data, as shown in Example 12-24. The **scan()** method is useful for debugging purposes as it allows you to take relevant actions in the code when a certain type of data is presented in YAML.

Example 12-24 *Using the* ***yaml.scan()*** *Method*

```
network-devices:
  routers:
  - rtr-ios-R1
  - rtr-xr-R2
  - rtr-nx-R3
  switches:
  - sw-ios-SW1
  - sw-xe-SW2
  - sw-nx-SW3
  servers:
  - srvr1: "linux"
    patched: True
    OS: CentOS
  - srvr2: "Microsoft Windows"
    patched: False
    OS: "Windows 10"
```

```
import yaml


with open('data.yaml') as f:
    data = yaml.scan(f, Loader=yaml.FullLoader)
    for tkn in data:
        print(tkn)
```

```
[root@node1 demo]# python3 demo.py
StreamStartToken(encoding=None)
BlockMappingStartToken()
KeyToken()
ScalarToken(plain=True, style=None, value='network-devices')
ValueToken()
BlockMappingStartToken()
KeyToken()
ScalarToken(plain=True, style=None, value='routers')
ValueToken()
BlockEntryToken()
ScalarToken(plain=True, style=None, value='rtr-ios-R1')
BlockEntryToken()
ScalarToken(plain=True, style=None, value='rtr-xr-R2')
BlockEntryToken()
ScalarToken(plain=True, style=None, value='rtr-nx-R3')
KeyToken()
ScalarToken(plain=True, style=None, value='switches')
ValueToken()
BlockEntryToken()
ScalarToken(plain=True, style=None, value='sw-ios-SW1')
```

```
BlockEntryToken()
ScalarToken(plain=True, style=None, value='sw-xe-SW2')
BlockEntryToken()
ScalarToken(plain=True, style=None, value='sw-nx-SW3')
KeyToken()
ScalarToken(plain=True, style=None, value='servers')
ValueToken()
BlockEntryToken()
BlockMappingStartToken()
KeyToken()
ScalarToken(plain=True, style=None, value='srvr1')
ValueToken()
ScalarToken(plain=False, style='"', value='linux')
KeyToken()
ScalarToken(plain=True, style=None, value='patched')
ValueToken()
ScalarToken(plain=True, style=None, value='True')
KeyToken()
ScalarToken(plain=True, style=None, value='OS')
ValueToken()
ScalarToken(plain=True, style=None, value='CentOS')
BlockEndToken()
BlockEntryToken()
BlockMappingStartToken()
KeyToken()
ScalarToken(plain=True, style=None, value='srvr2')
ValueToken()
ScalarToken(plain=False, style='"', value='Microsoft Windows')
KeyToken()
ScalarToken(plain=True, style=None, value='patched')
ValueToken()
ScalarToken(plain=True, style=None, value='False')
KeyToken()
ScalarToken(plain=True, style=None, value='OS')
ValueToken()
ScalarToken(plain=False, style='"', value='Windows 10')
BlockEndToken()
BlockEndToken()
BlockEndToken()
StreamEndToken()
```

As you have learned in this chapter, YAML is commonly used to build configuration files. These configuration files can be used along with Jinja templates to render configurations for various networking devices. Example 12-25 shows a BGP configuration. In this example, the data.yaml file has a number of BGP-related configurations, such as router ID, network advertisements, and neighbors. This data is rendered into a proper BGP configuration using Jinja templates. Building any configuration using YAML and Jinja involves three basic steps:

Step 1. Load the YAML data.

Step 2. Set the directory path and select the Jinja template.

Step 3. Render the data into the Jinja template to generate the configuration.

Example 12-25 demonstrates this process of generating a configuration.

Example 12-25 *Building a Configuration Using YAML and Jinja Templates*

```
! data.yaml

---

bgp:

  id: 100

  router_id: 1.1.1.1

  networks:

    - 192.168.1.1/32

    - 192.168.12.0/24

    - 192.168.13.0/24

  neighbors:

    - id: 2.2.2.2

      remote_as: 200

      afi: "ipv4 unicast"

    - id: 3.3.3.3

      remote_as: 300

      afi: "ipv4 unicast"
```

```
! templates/bgp_template.j2

router bgp {{ bgp.id }}

router-id {{ bgp.router_id }}

address-family ipv4 unicast

{% for network in bgp.networks %}

network {{ network }}

{% endfor %}

exit address-family

{% for neighbor in bgp.neighbors %}

neighbor {{ neighbor.id }}

  remote-as {{ neighbor.remote_as }}

  address-family {{ neighbor.afi }}

  exit

exit

{% endfor %}
```

```
import yaml

from jinja2 import Environment, FileSystemLoader


config_data = yaml.load(open('data.yaml'), Loader=yaml.FullLoader)

env = Environment(loader = FileSystemLoader('./templates'), trim_blocks=True,

lstrip_blocks=True)


template = env.get_template('bgp_template.j2')

print(template.render(config_data))
```

```
[root@node1 demo]# python3 demo.py
```

```
router bgp 100
router-id 1.1.1.1
address ipv4 unicast
network 192.168.1.1/32
network 192.168.12.0/24
network 192.168.13.0/24
exit address-family
neighbor 2.2.2.2
  remote-as 200
  address-family ipv4 unicast
  exit
exit
neighbor 3.3.3.3
  remote-as 300
  address-family ipv4 unicast
  exit
exit
```

Note

Chapter 6, "Python Applications," further discusses working with YAML data and Jinja templates as well as other Python libraries, such as NAPALM and Nornir.

Summary

YAML is a human-readable data serialization language used to build configurations. YAML is also a superset of JSON, and so there are many similarities between JSON and YAML. YAML allows you to represent data in three different formats:

- Scalars
- Sequence
- Mapping

In this chapter, you have learned that YAML represents native data structures using nodes and tags. Unlike other formats, such as JSON, YAML has supports comments. YAML allows you to represent common data structures, such as strings, integers, floating points, and Booleans, as well as complex data structures. YAML supports anchors and merge keys, which help with duplication of data as well as inheritance.

The Python PyYAML package allows programmers to serialize Python objects into a YAML stream by using the **yaml.dump()** method and a YAML stream to a Python object by using the **yaml.load()** method. Programmers and network automation engineers can also use Jinja templates along with the PyYAML package to build configurations for network devices.

Part VI: Modeling

Chapter 13. YANG

This chapter discusses data modeling in general and the YANG modeling language in particular. It's very important to understand these topics, as YANG data models are very important in network automation today.

A Data Modeling Primer

Before you delve into the details of YANG, it is important that you understand the problem it attempts to solve. YANG provides a framework for *modeling* (or *structuring*) data. The following sections cover the difference between structured and unstructured data, what a data model is, why we need data models, and the problems that data modeling solves.

What Is a Data Model?

Information is everywhere. It is not an exaggeration to say that we are surrounded by various types of data all the time. Even now, you are consuming data by reading this book. If the information around you isn't structured somehow, efficient consumption and understanding of this data will be a problem. For example, the information in this book isn't related to the weather outside, or the currency exchange rate in the foreign exchange market. But this is not a problem, because different data—data in the book, data about the weather, data about the exchange rate—exists *in different contexts*.

Say that there was a mistake in this book such that one page was printed in English, another in Chinese, and a third in an unreadable font, such as Wingding. In such a case, the data would not be structured properly within *a single context*. In order to avoid such chaos, this book is created following certain rules:

- It has a clear structure defined in the table of contents at the beginning.
- All the chapters and paragraphs follow that predefined structure, without deviation.
- All the chapters are written using the same language and writing style.
- Each chapter has the same internal structure as others, starting with the introduction and ending with a summary.
- Information representation —such as fonts, font sizes, images, tables, and examples—is the same throughout the book.

Just think about this list of constraints for a moment. There is not a single word about the information that is actually contained in the book. There are only rules (constraints) and characteristics regarding *how*the information in this book *should* be composed. And that is the key, because any *data model* is a set of rules and characteristics that define how the data should be structured, formatted, and represented in order to be efficiently consumed. *Data modeling* is the process of defining those rules and characteristics. It's legitimate to say that for this book, the data modeling is done by Cisco Press; the team at Cisco Press defines the formatting, prepares the templates, and approves the table of contents before the book is written. Then, before the book is sent to print, the team reviews the content to make sure that the rules are being followed at every step of the process and that the final printed book conforms to those rules.

Another example of a data model is an application form you need to fill out to get a visa before visiting some exotic country. It provides a description of the information required from you (such as your name, surname, and birthday) and in which format each field is to be provided (for example, letters, numbers, photo). But the application form doesn't initially contain the actual data; you who populates this data by filling in the form. On the other hand, the data modeling for the application form is done by the government (or its subcontractors) of the country to which you are applying for a visa.

We could list examples of data models indefinitely, but the general idea of what a data model is should be clear by now.

Why Data Modeling Matters

Consider again the visa application example from the previous section. Say that after you have completed the visa application, the employees of the appropriate government service process your data. The first things they check are the *completeness* and *format* of the information provided—whether all the mandatory fields are filled in and whether all the relevant additional documents are attached, in the formats requested. The same set of data must be provided by all applicants. If the information isn't complete or if it is provided in the wrong format, the application is rejected because the data cannot be validated. If the information is complete and provided in the correct format, it's processed further in order to make a decision on granting or rejecting the visa.

As you can see from the visa application example, data models help transform informational chaos into structured data. They set explicit rules on what data should be provided by the transmitter, and in which format should the data be provided. They also set clear expectations for the receiver. If there were not such clear rules, the communication of the information and its further processing would be much more complicated—if were possible at all. Therefore, data models play a crucial role in our everyday lives, even if we don't realize that we are using them.

Now that you have some familiarity with the concept of data modeling and its applicability in general, it's time to start looking at these concepts from the perspective of network configuration and management.

Let's consider the process of provisioning of an IP VPN service for a customer. What information do you need to provision it? The answers to this question will determine the data model for the provisioning of the IP VPN service. You might want to structure the data model in the following way:

- **Field 1**: endpoint A; format: text; description: router hostname at the 1st site
- **Field 2**: port A; format: text; description: customer's attachment circuit for the 1st site
- **Field 3**: IP address A; format: numbers and special characters [.:/]; description: IP address on the customer-facing port at the 1st site
- **Field 4**: endpoint B; format: text; description: router hostname at the 2nd site
- **Field 5**: port B; format: text; description: customer's attachment circuit for the 2nd site
- **Field 6**: IP address B; format: numbers and special characters [.:/]; description: IP address on the customer-facing port at the 2nd site
- **Field 7**: RD; format: numbers and special characters [.:]; description: Route Distinguisher for the customer service
- **Field 8**: Import RTs; format: numbers and special characters [.:]; description: Import Route Targets for the customer service
- **Field 9**: Export RTs; format: numbers and special characters [.:]; description: Export Route Targets for the customer service

There might be many more optional parameters, but these nine are minimally required to establish the connectivity for a customer. (This example is covered in more detail in Chapter 11, "JSON and JSD," but for the purpose of this chapter, this high-level description is enough.)

So, these nine fields constitute your data model for the IP VPN service.

Once you have a data model, you need to consider how this data model should be expressed or documented for effective use. You could create your own syntax and keywords, but you need to make sure that other applications or devices understand and support that syntax. Basically, you need a data modeling language, and it must satisfy at least the following categories:

• It should be clear and consistent in terms of syntax and keywords to avoid ambiguity.

• It should provide the possibility to create any data model for any use case.

• It should be flexible enough to extend when necessary.

• It should be product, platform, and vendor agnostic.

All these constraints are important, but the last point is of particular importance. For many years, customers were locked to certain vendor operating system implementations and had to master CLI skills for different platforms. In the era of automation and programmability, network devices cannot be operated exclusively through a CLI, especially when it comes to service provisioning/maintenance. They should be operated using standardized APIs using the same data model—and the industry is developing in that direction. For now, at least the language that is used to describe data models (and data modeling) should be the same across the industry. Thanks to collaboration within the network industry, led by the IETF, such language exists, and it's called YANG.

YANG Data Models

You can find the original description of the YANG language in RFC 6020, "YANG: A Data Modeling Language for the Network Configuration Protocol (NETCONF)." Since that RFC was released, the networking community, led by the IETF, has significantly developed YANG, and today it is considered to be a generic data modeling language that can be used in conjunction with any type of protocol used for programming network elements, the most popular being NETCONF, RESTCONF, and gRPC. The more recent RFC 7950, "The YANG 1.1 Data Modeling Language," covers YANG Version 1.1.

So far, YANG has proven to be a sweeping success, and a number of factors have contributed to this success. First of all, YANG syntax is quite straightforward and easy to understand. The second important factor is that YANG is strongly supported by the telecommunications industry; major vendors, standardization bodies (including the Internet Engineering Task Force [IETF], Metro Ethernet Forum [MEF], and Broadband Forum [BBF]), and informal communities (including OpenConfig, a vendor-neutral working group driven by Google) are using YANG extensively to create their data models. And the third important factor is that YANG is a language, which means you can create your own YANG modules based on application requirements.

Let's consider an example that emphasizes the importance of the YANG language, particularly for data modeling and network programmability. You are familiar with the process of configuring network devices (such as routers, switches, and firewalls). You can may think about the functions of those network devices, referred to as network functions (NFs), as lines of code performing different actions. These lines of configuration are basically how certain parameters (for example, IP addresses or VLAN numbers alongside encapsulation types) are mapped to certain objects (such as interfaces). In a nutshell, a data model shows the relationship between parameters and different objects. YANG provides a simple and powerful way to define such a relationship.

A *YANG module* is a tree-like structure that describes some information in a hierarchical way. The smallest component of this tree model is a single parameter in key/value format, where the key is the specific parameter, and the value is some information that describes the quality or quantity of the parameter. Going from leafs of the tree to branches and further up shows the logic involved in aggregating the smallest and most explicit components of the information tree together to describe more complex objects. (*Smallest* here refers to the fact that each of these components cannot be broken down further into constituent components.) For instance, if we think again about the earlier example of the data model for the router interface, these would be the smallest components of the information tree:

• IPv4 address

• IPv4 prefix length

• IPv6 address

• IPv6 prefix length

• Interface type

• Interface name

• Maximum transmission unit (MTU)

• Encapsulation type

• (Encapsulation) VLAN ID

This information can be structured in a hierarchical format to represent the data model of the interface, as shown on the Figure 13-1.

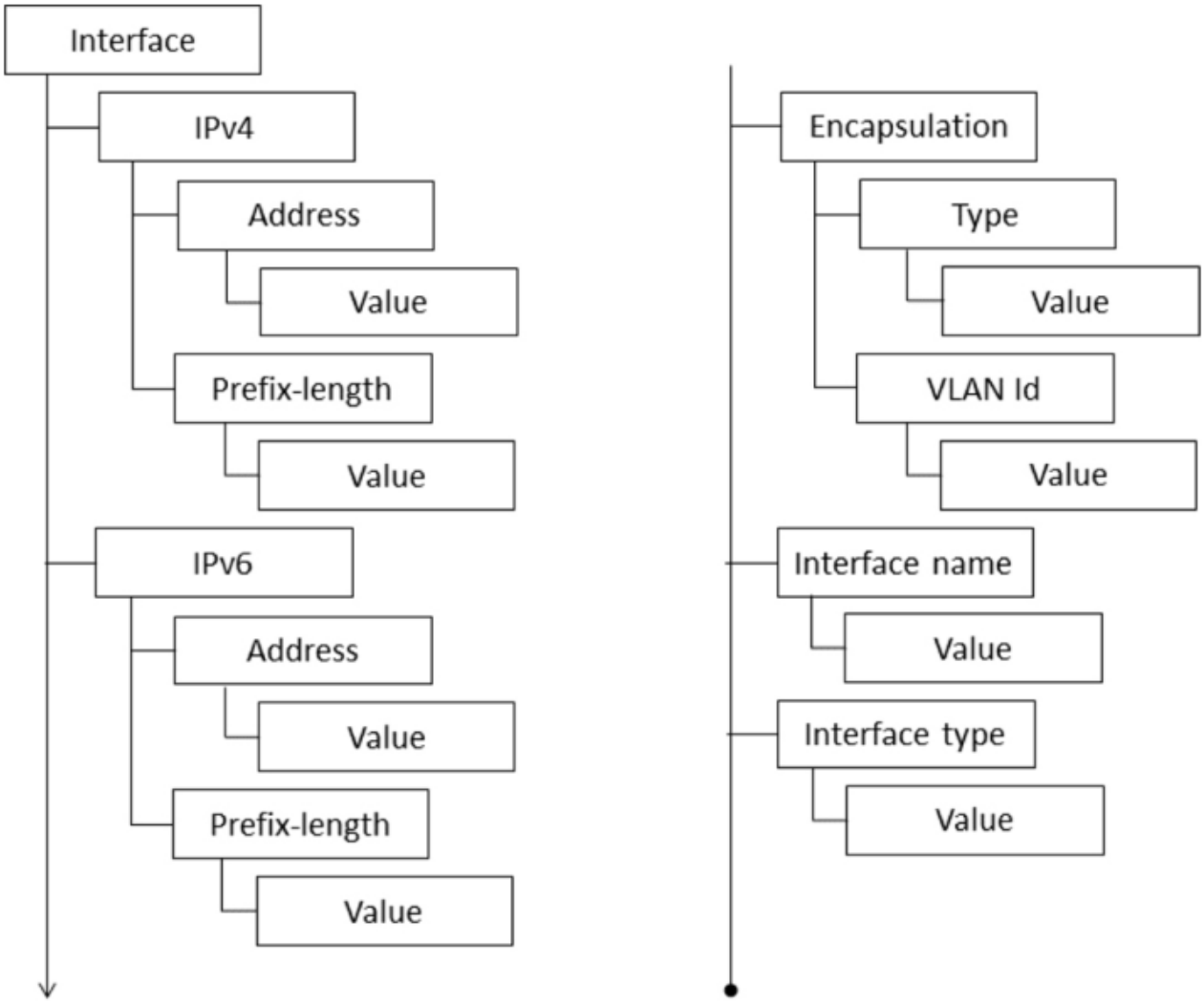

Figure 13-1 *Generic Overviewof a YANG Module*

The protocols used in network programmability don't use YANG directly; rather, they use representations of YANG data models in JSON or XML format. Later in the book, you will encounter real-life scenarios of the use of YANG data models for programming network functions on Cisco IOS XE, IOS XR, and NX-OS, as well as several other non-Cisco platforms.

Structure of a YANG Module

To help you better understand YANG data modeling in the context of network programmability, this section uses examples of real YANG modules for Cisco IOS XR. To start the journey into the YANG world, first of all, you need to understand the structure of a YANG module, which has three major parts:

• Module header statements

• Revision statements

• Definition statements

The sequence of these parts is important; the module header statement must appear first in a module, and the definition statements must appear last.

Module header statements contain important information about the module in general, such as the name of the module, the XML namespace used, and additional dependencies on external YANG modules needed for the operation of the module, if any. One of the very useful features of YANG is that it's possible to have all information related to the data model in a single module, and it's also possible to split this information across several modules and reuse modules, where necessary. Example 13-1 shows the module header statements from a Cisco YANG module. Generally, each vendor has its own YANG modules (for example, Cisco, Juniper, Nokia). Further, for a vendor, there might be different flavors of operating systems, such Cisco IOS XE, IOS XR, and NX-OS, and each of these operating system flavors has its own modules as well. Finally, modules can be changed between the releases; therefore, it is very important to check your network operating system type and version before starting to work with YANG modules.

Example 13-1 *Module Header Statements in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ifmgr-cfg.yang


 module Cisco-IOS-XR-ifmgr-cfg {


  /*** NAMESPACE / PREFIX DEFINITION ***/

  namespace "http://cisco.com/ns/yang/Cisco-IOS-XR-ifmgr-cfg";

  prefix "ifmgr-cfg";


  /*** LINKAGE (IMPORTS / INCLUDES) ***/
```

```
  import Cisco-IOS-XR-types { prefix "xr"; }


  /*** META INFORMATION ***/

  organization "Cisco Systems, Inc.";

  contact

    "Cisco Systems, Inc.

     Customer Service

     Postal: 170 West Tasman Drive

     San Jose, CA 95134

     Tel: +1 800 553-NETS

     E-mail: cs-yang@cisco.com";


  description

    "This module contains a collection of YANG definitions

     for Cisco IOS-XR ifmgr package configuration.

     This module contains definitions

     for the following management objects:

       global-interface-configuration: Global scoped configuration

         for interfaces

       interface-configurations: interface configurations

     Copyright (c) 2013-2016 by Cisco Systems, Inc.

     All rights reserved.";


------ The output is truncated for brevity ------
```

Much like other programming languages, and as you have seen with Bash and Python in earlier chapters, YANG provides the facility to insert comments in a YANG module. These comments are enclosed between slash characters, like this: */*** *Comment text****/*. Notice in Example 13-1 that Cisco uses comments extensively to provide useful hints to module users.

The first line of Example 13-1 contains the keyword **module** to define the name of the module. In Chapter 14, "NETCONF and RESTCONF," you will learn how to get a list of all supported YANG modules on any device that is YANG capable. The common practice is to have the value with **module** exactly the same as the YANG filename. The next statement defines the namespace, which is how this particular YANG module can be called in XML-encoded messages. The **prefix** statement defines the string that is used as a prefix to access the module or to refer to definitions contained in the module. XML and XML schemas are covered in detail in Chapter 10, "XML and XSD." The **import** statement points to the other YANG modules that are required for the proper operation of the module containing the **import** statement(s). Typically, it's convenient to define some data types or groupings once and then import them to all other YANG modules that require such information. You will learn details about both data types and groupings later in this chapter. The rest of the information in the module header statements is meta-information that explains what organization released this module, how to contact the organization, and a description of the module content.

The next sections of Example 13-1, each of which starts with a **revision** statement, contains the history log of the YANG module development, which indicates which software release this particular YANG data model is applicable to and/or which changes are introduced in this version of the module compared to the previous version. You will find this information important when you try to apply the data model of one software version (such as Cisco IOS XR 5.3.1) to a device running another software version (such as Cisco IOS XR 6.4.1). Example 13-2 provides an example of the changes applied to one of the Cisco YANG modules.

Example 13-2 ***revision*** *Statements in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ifmgr-cfg.yang

! The output is truncated for brevity


  revision "2015-07-30" {

    description

      "Descriptions updated.";

  }


  revision "2015-01-07" {

    description

      "IOS XR 5.3.1 revision.";

  }


! The output is truncated for brevity
```

As you can see, the structure of this section is self-explanatory. Each **revision** statement is named with a date indicating when the YANG module was changed and contains a **description** field that shows an explanation of the modifications applied. There might be multiple such **revision** entries if there were several changes involved in developing this YANG module. It isn't mandatory to update these statements, but doing so is a good practice, as it makes life easier for those who are using the YANG module for configuration of network functions, as it helps

them know what exactly they need to check within an updated YANG module.

Data Types in a YANG Module

Data types are the fuel on which the YANG data model engine runs. A definition of a data model comprises specific data that is expressed in certain data types and structured in a certain hierarchy. RFC 6020, which defines YANG, indicates that YANG has built-in data types and also derived data types.

Built-in Data Types

Built-in data types are types that exist in the YANG language by default and are available in all the YANG modules. You can use built-in data types in any of your models. Table 13-1 lists some of the most commonly used built-in data types to help you get a feel for what a data type is. The full list of built-in data types is available in RFC 7950.

Table 13-1 *Built-in Data Type in YANG*

Data Type	Description
binary	Any binary data
Boolean	**True** or **false**
empty	A leaf that does not have any value
enum	An enumerated string that provides a list of all allowed values for a variable (much like a drop-down menu with predefined values)
int8	An 8-bit signed integer
int16	A 16-bit signed integer
string	A human-readable string
uint8	An 8-bit unsigned integer (The difference between an unsigned integer and a signed one is that an unsigned integer cannot be negative. For example, int8 covers the range –127 to 127 (in decimal format), whereas unit8 covers the range 0 to 255.)
uint32	A 32-bit unsigned integer

To get a better understanding of how built-in data types are used, take a look at Example 13-3.

Example 13-3 *Built-in Data Types in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ifmgr-cfg.yang

------ The output is omitted for brevity ------


      leaf bandwidth {
        type uint32 {
          range "0..4294967295";
        }
        units "kbit/s";
        description "The bandwidth of the interface in kbps";
      }
      leaf link-status {
        type empty;
        description
          "Enable interface and line-protocol state change
          alarms";
      }
      leaf description {
        type string;
        description "The description of this interface";
      }

------ The output is truncated for brevity ------
```

You will learn more about the keyword **leaf** later in this chapter. For now, you can just think of it as an arbitrary parameter expressed in the data type defined by the value of the keyword **type**. In addition, you can configure the range of the allowed values for this parameter by using the keyword **range**.

Derived Data Types

Despite the large number of built-in data types, you may at times need to create custom data types, referred to as derived data types. A derived data type is available only within the particular YANG module where it is defined or in any other YANG module that imports the YANG module where this data type is defined. Example 13-4 shows a derived data type defined using a built-in type.

Example 13-4 *A Derived Data Type in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ifmgr-cfg.yang

! The output is truncated for brevity
```

```
  typedef Interface-mode-enum {
    type enumeration {
      enum default {
        value 0;
        description "Default Interface Mode";
      }
      enum point-to-point {
        value 1;
        description "Point-to-Point Interface Mode";
      }
      enum multipoint {
        value 2;
        description "Multipoint Interface Mode";
      }
      enum l2-transport {
        value 3;
        description "L2 Transport Interface Mode";
      }
    }
    description "Interface mode enum";
  }


------ The output is truncated for brevity ------
```

They keyword **typedef** instructs the YANG module to create a new derived data type. This new derived data type is using the built-in data type **enumeration**, which is described in Table 13-1 as the type for enumerated strings. Each **enum** statement has the name of one of these enumerated strings and contains an associated value and description. As you might have noticed, you can add a field description almost anywhere. Using description statements extensively throughout your YANG module enhances its clarity to anyone who plans on using it. (Later in this chapter, in Example 13-6, you will see the usage of this new derived data type.)

Data Modeling Nodes

By this point, you should be familiar with the structure of a YANG module and the associated data types. The next logical step is to start building a data model. The building blocks of a data model are called *nodes*. Each node represents some object, and each node has a name and either a value or a set of child nodes. A YANG module simply defines a hierarchy of nodes and the interaction between those nodes. As per RFC 7950, Section 4.2.2, YANG defines four node types:

- Leaf nodes
- Leaf-list nodes
- Container nodes
- List nodes

The following sections discuss these node types.

Leaf Nodes

A leaf node, which is the smallest building block of a YANG data model, contains data of one particular type (for example, string) and has no possibility to nest any further data inside it. You will find a lot of leaf nodes of different data types in any YANG module. Example 13-5 shows the leaf nodes in a Cisco YANG module.

Example 13-5 *Leaf Nodes with Built-in Data Types in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ifmgr-cfg.yang
------ The output is omitted for brevity ------


      leaf bandwidth {
        type uint32 {
          range "0..4294967295";
        }
        units "kbit/s";
        description "The bandwidth of the interface in kbps";
      }
      leaf link-status {
        type empty;
```

```
        description
          "Enable interface and line-protocol state change
          alarms";
      }
      leaf description {
        type string;
        description "The description of this interface";
      }
------ The output is truncated for brevity ------
```

The keyword **leaf** defines the type of the node. A node always has a name; for example, in Example 13-5, the nodes are named **bandwidth**, **link-status**, and **description**. These names are used when you construct XML or JSON messages (as you will learn in the following chapters). Inside a **leaf** entry, you use the **type** keyword to define the data type used. This is the only mandatory parameter that must be included in all leaf nodes; the rest of the parameters are optional. So, you can optionally define the field **units**, which provides information about units of measurement for this entry, **description**, and many other parameters based on the needs of the data model.

Note

Refer to RFC 7950 for a full list of the parameters defined for YANG.

Example 13-5 shows the usage of built-in data types, but there is no difference in how derived data types are used. Example 13-6 shows how the derived data type defined in Example 13-4 is used in a leaf node.

Example 13-6 *Leaf Nodes with Derived Data Type in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ifmgr-cfg.yang
------ The output is omitted for brevity ------

      leaf interface-mode-non-physical {
        type Interface-mode-enum;
        default "default";
        description
          "The mode in which an interface is running. The
          existence of this object causes the creation of
          the software subinterface.";
      }
------ The output is truncated for brevity ------
```

Besides the derived data type **interface-mode-enum**, Example 13-6 shows another interesting piece of information. The keyword **default** allows you to define the default value for this parameter in case you don't define it explicitly. It makes sense to have this type of protection so that the application using this YANG module doesn't crash if you forget to specify a mandatory parameter. In Example 13-6, for example, the network device running the Cisco IOS XR SW will automatically set the interfaces type to the value defined in the keyword **default** if it is not explicitly specified.

Leaf-List Nodes

In certain cases, where a data model requires several values of a single parameter, you need to define several leaf nodes of the same type. For such a scenario, you use the leaf-list node, as shown in Example 13-7.

Example 13-7 *Leaf-List Nodes in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-infra-policymgr-cfg.yang
 ------ The output is omitted for brevity ------

     leaf-list ipv4-dscp {
       type Dscp-range;
       max-elements 8;
       description "Match IPv4 DSCP.";
     }
     leaf-list ipv6-dscp {
       type Dscp-range;
       max-elements 8;
       description "Match IPv6 DSCP.";
     }
     leaf-list dscp {
       type Dscp-range;
```

```
        max-elements 8;
        description "Match DSCP.";
      }
------ The output is truncated for brevity ------
```

As shown in Example 13-7, the keyword **leaf-list** creates an entry for a leaf-list node. You use the field **type** to define the data type, exactly as you do for a leaf node. You might also want to add the optional field **max-elements**, which limits the number of leaf values in this leaf list.

Note

For a comprehensive list of optional parameters available for leaf and leaf-list nodes, refer to RFC 7950.

Container Nodes

The third node type available in YANG is a container node. It doesn't have any value itself; rather, it contains any number of other nodes of any type (leaf, leaf-list, container, or list). So, if you need to group different sets of parameters related to the same object, a container node is a natural choice. Example 13-8 shows several parameters being defined simultaneously for one object.

Example 13-8 *A Container Node in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ifmgr-cfg.yang
------ The output is omitted for brevity ------


      container encapsulation {
        description "The encapsulation on the interface";
        leaf encapsulation {
          type string;
          description "The encapsulation - e.g. hdlc, ppp";
        }
        leaf capsulation-options {
          type int32;
          description
            "The options for this capsulation, usually '0'";
        }
      }


------ The output is truncated for brevity ------
```

You define a container node by using the keyword **container** followed by the name of the node. As you can see in Example 13-8, the only field that is defined directly within the container is **description**, and the rest are **leaf** entries. The container node could also nest another container, as there is no limit on the number of nested elements and levels of nesting, as shown in Example 13-9.

Example 13-9 *Container Node with a Nested Container in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ipv4-io-cfg.yang
------ The output is omitted for brevity ------


    container ipv4-network {
      description
        "Interface IPv4 Network configuration data";
      container bgp-pa {
        description
          "Interface ipv4 bgp configuration";
        container input {
          description
            "Input";
          leaf source-accounting {
            type boolean;
            description
              "BGP PA configuration on source";
          }


------ The output is truncated for brevity ------
```

List Nodes

Like a container node, a list node can have any number of leaf, leaf-list, container, and list nodes nested inside. The key difference is that typically list nodes are used when you need to define a similar set of parameters of many types, as in the case of configuring network interfaces or BGP neighbors. In addition, in the vast majority of cases, a list node has at least one leaf entry as a key (although this is not mandatory), and this is significant for building a tree. (See RFC 7950 Section 7.8 for further information on this particular point.) Example 13-10 shows an example of the structure of a list node.

Example 13-10 *A List Node in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ifmgr-cfg.yang

------ The output is omitted for brevity ------

        list mtu {

          key "owner";

          description "The MTU for the interface";

          leaf owner {

            type xr:Cisco-ios-xr-string;

            description

              "The Owner of the interface - eg. for

              'LoopbackX' main interface this is 'loopback'";

          }

          leaf mtu {

            type uint32 {

              range "64..65535";

            }

            mandatory true;

            description "The MTU value";

          }

        }

------ The output is truncated for brevity ------
```

As shown in Example 13-10, this node type is created by using the keyword **list** followed by the name. The next field, **key**, is a mandatory attribute. The key is a leaf node that must have an explicit value and that uniquely identifies an entry inside the list. The leaf identified as the key cannot have a default value since a default value would defy the whole purpose of having a unique identifier for each list entry. In addition, the optional attribute **mandatory** is used to specify that a particular leaf node must be configured.

Grouping Nodes

In some situations, you might need to use the same set of parameters (or even just one parameter) in different contexts. To achieve that, you can use a YANG construct called *grouping*. After you define a grouping, you can call it many times wherever you need it. In Example 13-11, the grouping **BFD** is created and then instantiated later in the module, under the container **bgp**. The **description** statement for the grouping indicates what this grouping accomplishes.

Example 13-11 *Reusable Node Group in a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ipv4-bgp-cfg.yang

------ The output is omitted for brevity ------


  grouping BFD {

    description "Common node of global, vrf-global";


    container bfd {

      description "BFD configuration";

      leaf detection-multiplier {

        type uint32 {

          range "2..16";

        }

        description

          "Detection multiplier for BFD sessions created

          by BGP";

      }

      leaf interval {

        type uint32 {

          range "3..30000";
```

```
        }
        units "millisecond";
        description
          "Hello interval for BFD sessions created by BGP";
      }
    }
  }

------ The output is omitted for brevity ------

  container bgp {
    description "BGP configuration commands";

    list instance {
      list instance-as {
        list four-byte-as {
          container vrfs {
            list vrf {
              container vrf-global {
                uses BFD;
              }
            }
          }
        }
      }
    }
------ The output is truncated for brevity ------
```

Following the logic of strongly typed programming languages, any type or class of data should be defined prior to being used. You therefore create a grouping right after defining the derived data types at the beginning of a YANG module. Moreover, it's much easier to read the YANG module if you follow this convention, as you can easily follow the sequence of definitions and data usage.

You use the keyword **grouping** followed by the name of a grouping to create a context that groups a set of other node types. Later on, when you need to call a grouping of the nodes in any other context, you use the keyword **uses** followed by the name of the grouping. The **uses** statement is used to reference a grouping definition. It takes one argument, which is the name of the grouping.

Augmentations in YANG Modules

By now you should be familiar with the four YANG node types—leaf, leaf-list, container, and list—as well as with groupings. Keep in mind that for the purpose of this section, the term *node* refers to one of the four YANG node types, not to a network element itself.

The YANG language allows you to extend a parent YANG data model with additional nodes. To understand where this may be applicable, consider an interface IPv4 address: You need to configure an IP address only if the interface is active and working in routed mode, and you do not need to configure it in other cases (for example, if the interface is working in switching mode). In other words, the node representing the interface will always exist, regardless of its contents, whereas the node representing the IPv4 address may or may not exist within the interface node. This an approach is called *augmentation* in the YANG language. Example 13-12 illustrates the augmentation process for the interface IPv4 address use case.

Example 13-12 *Augmenting a Cisco IOS XR YANG Module*

```
$ cat Cisco-IOS-XR-ipv4-io-cfg.yang

  /*** NAMESPACE / PREFIX DEFINITION ***/
  namespace "http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-io-cfg";
  prefix "ipv4-io-cfg";

  /*** LINKAGE (IMPORTS / INCLUDES) ***/
  import ietf-inet-types { prefix "inet"; }
  import Cisco-IOS-XR-types { prefix "xr"; }
  import Cisco-IOS-XR-ifmgr-cfg { prefix "a1"; }

------ The output is omitted for brevity ------
  augment "/a1:interface-configurations/a1:interface-configuration" {
    container ipv4-network {
```

```
description "Interface IPv4 Network configuration data";

container addresses {
  description "Set the IP address of an interface";

  container secondaries {
    description "Specify a secondary address";
    list secondary {
      key "address";
      description "IP address and Mask";
      leaf address {
        type inet:ipv4-address-no-zone;
        description "Secondary IP address";
      }
      leaf netmask {
        type inet:ipv4-address-no-zone;
        mandatory true;
        description "Netmask";
      }
      leaf route-tag {
        type uint32 {
          range "1..4294967295";
        }
        description "RouteTag";
      }
    }
  }

  container primary {
    presence "Indicates a primary node is configured.";
    description "IP address and Mask";
    leaf address {
      type inet:ipv4-address-no-zone;
      mandatory true;
      description "IP address";
    }
    leaf netmask {
      type inet:ipv4-address-no-zone;
      mandatory true;
      description "Netmask";
    }
  }
  leaf unnumbered {
    type xr:Interface-name;
    description
      "Enable IP processing without an explicit
      address";
  }
  leaf dhcp {
    type empty;
    description "IPv4 address and Mask negotiated via DHCP";
  }
}
```

```
    }
  }
------ The output is truncated for brevity ------
```

The **augment** keyword followed by the name of a YANG node augments, or adds, that node to the primary module. Pay attention to the name of the augmented node, which starts with **a1**. In the **LINKAGE** section in Example 13-12, you can see the module header statements to import other YANG modules. Each of the imported modules is associated with some prefix, which is used to reference the particular YANG module within the body of the current YANG module. Following this logic, the line **import Cisco-IOS-XR-ifmgr-cfg { prefix "a1"; }** associates the imported module (**Cisco-IOS-XR-ifmgr-cfg**) with the prefix **a1**, which is then prepended (along with a colon) to the name of the augmented node (that is, **/a1:interface-configurations/a1:interface-configuration**). **augment** explicitly knows which YANG node or nodes it must extend.

The crucial point here is that an *augmented* YANG module has no knowledge that is has been augmented. This information lies solely with the *augmenting* module. The graphical representation in the Figure 13-2 illustrates how augmented and augmenting YANG modules are connected to each other.

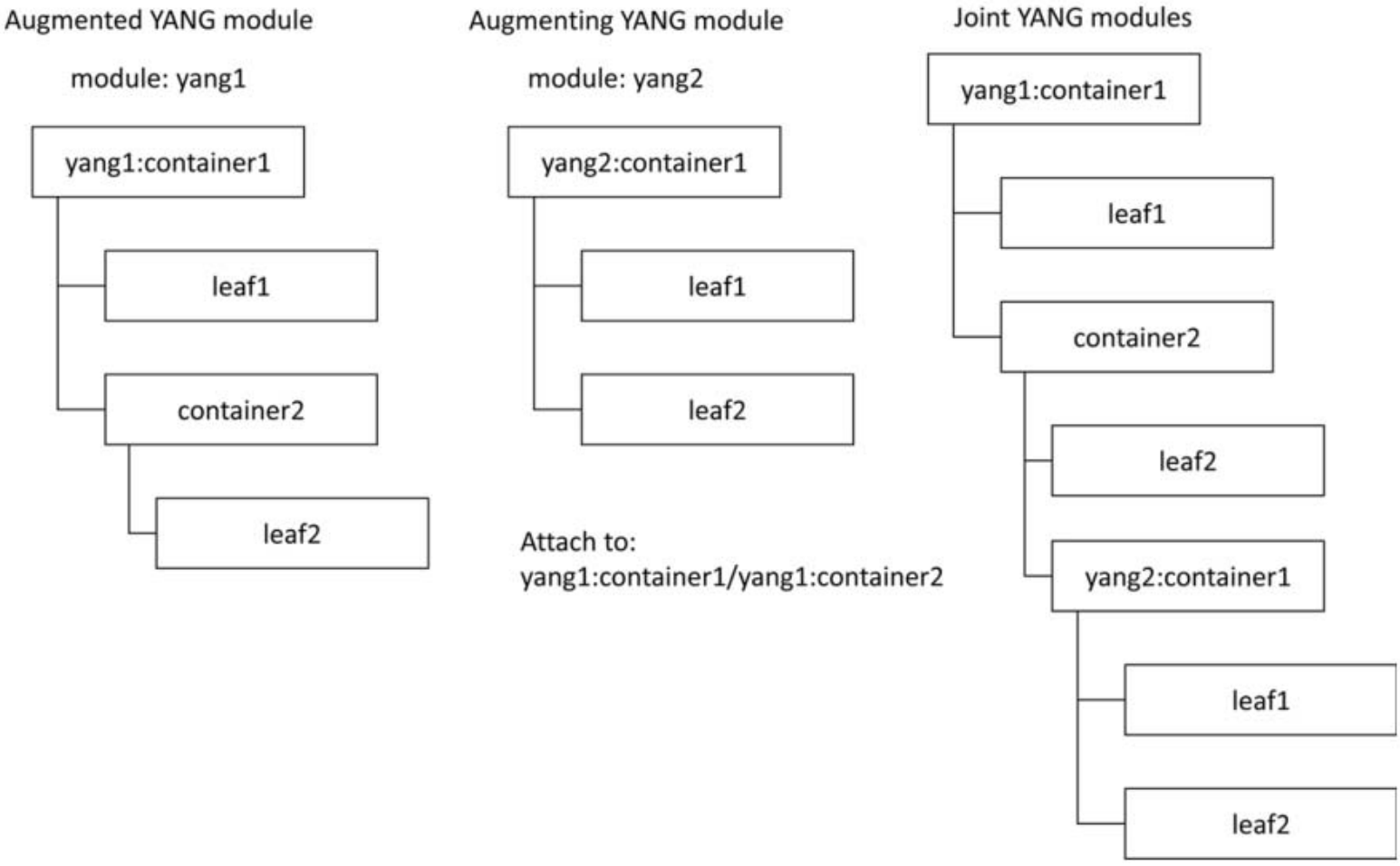

Figure 13-2 *YANG Augmentation Example*

Deviations in YANG Modules

So far in this chapter, the discussion has been about how you can create a YANG data model or extend its nodes through augmentation. Nevertheless, sometimes it might be necessary to tailor the scope of the initial YANG data models or to set some limits for just some cases or network functions.

For example, Cisco uses not only self-developed YANG models, but also third-party ones that are either standardized IETF YANG modules or community-driven OpenConfig YANG modules. You will learn shortly details about each of them, but for now, you can think about them as just other data models. The key point is that a vendor's adaptation of a third-party YANG model (such as an IETF model) into a particular operating system version could deviate from the original representation. Another reason for deviations might be the situation where a YANG model is developed in general for a software type and version (such as Cisco IOS XR 6.4.1 or Cisco IOS XE 16.9.1), but each network hardware family has a different set of capabilities and supported protocols (such as Cisco NCS 5x00 versus ASR 9000 for IOS XR and Cisco ASR 1000 versus Cisco Catalyst 3850 for IOS XE). In such a case, the YANG data model must be adapted to match the capabilities of a particular platform.

The following list sums up the major reasons a YANG model might deviate from the original representation:

- A vendor doesn't implement in its software all the features defined in a third-party YANG model.
- A vendor has another internal data structure, and needed functions are implemented via augments somewhere else.
- A vendor references a generic YANG data model for a software type but needs to adapt it to a particular hardware platform family.

To avoid any misbehavior in all cases, you need to use the concept of deviation in the YANG language. From a structure perspective, the deviations are YANG modules, which have exactly the same structure as any other YANG modules, but these YANG modules have specific keys that explain what is not supported in particular implementations. Example 13-13 shows a list of deviations from the original OpenConfig and IETF YANG modules in Cisco IOS XE 16.9.1.

Example 13-13 *Deviations from the YANG Modules for Cisco IOS XE 16.9.1*

```
Cisco-IOS-XE-switch-deviation.yang

cisco-xe-ietf-event-notifications-deviation.yang
```

```
cisco-xe-ietf-ip-deviation.yang
cisco-xe-ietf-ipv4-unicast-routing-deviation.yang
cisco-xe-ietf-ipv6-unicast-routing-deviation.yang
cisco-xe-ietf-ospf-deviation.yang
cisco-xe-ietf-routing-deviation.yang
cisco-xe-ietf-yang-push-deviation.yang
cisco-xe-openconfig-acl-deviation.yang
cisco-xe-openconfig-bgp-deviation.yang
cisco-xe-openconfig-bgp-policy-deviation.yang
cisco-xe-openconfig-if-ethernet-deviation.yang
cisco-xe-openconfig-if-ip-deviation.yang
cisco-xe-openconfig-if-poe-deviation.yang
cisco-xe-openconfig-interfaces-deviation.yang
cisco-xe-openconfig-interfaces-switching-deviation.yang
cisco-xe-openconfig-network-instance-deviation.yang
cisco-xe-openconfig-openflow-deviation.yang
cisco-xe-openconfig-platform-deviation.yang
cisco-xe-openconfig-routing-policy-deviation.yang
cisco-xe-openconfig-spanning-tree-deviation.yang
cisco-xe-openconfig-system-deviation.yang
cisco-xe-openconfig-vlan-deviation.yang
```

Later in this chapter, you will learn where you can get the YANG modules for Cisco and other vendors. For now, it's important to understand the structure of a YANG module with deviations and how the deviations are applied to the parent module. Example 13-14 shows the deviations based on one Cisco IOS XE module.

Example 13-14 *Deviations from the IETF IP Module in Cisco IOS XE 16.9.1 for non-supported leafs*

```
$ cat cisco-xe-ietf-ip-deviation.yang
module cisco-xe-ietf-ip-deviation {
  namespace "http://cisco.com/ns/cisco-xe-ietf-ip-deviation";

  prefix ip-devs;

  import ietf-interfaces {
     prefix if;
  }

  import ietf-ip {
     prefix ip;
  }

------ The output is truncated for brevity ------

  deviation /if:interfaces/if:interface/ip:ipv4/ip:enabled {
      deviate not-supported;
      description  "Not supported in IOS-XE 3.17 release.";
  }

  deviation /if:interfaces/if:interface/ip:ipv4/ip:forwarding {
      deviate not-supported;
      description  "Not supported in IOS-XE 3.17 release.";
  }

  deviation /if:interfaces/if:interface/ip:ipv4/ip:mtu {
      deviate not-supported;
      description  "Not supported in IOS-XE 3.17 release.";
```

```
  }


  deviation /if:interfaces/if:interface/ip:ipv4/ip:neighbor {

      deviate not-supported;

      description  "Not supported in IOS-XE 3.17 release.";

  }


------ The output is truncated for brevity ------
```

In Example 13-14, the string following the **deviation** keyword provides the path within the hierarchy of a specific YANG module where the behavior of that module, on a specific platform, will be different from that of the original module. In Example 13-14, you can see that different prefixes are used, which means the deviations are applied to augmented modules, which can't exist without parent modules. (For example, IP address augmentation can't exist outside the parent interface YANG module.)

Within the **deviation** entry is an instruction using the keyword **deviate** which states that a particular action is to be applied to the original YANG module (indicating how it should be changed). In Example 13-14, the action is **not-supported**, which tells you that this particular node isn't supported in Cisco IOS XE 16.9.1. There might be other actions; Example 13-15 shows one of them.

Example 13-15 *Deviations of the IETF IP Module in Cisco IOS XE 16.9.1 for modified leafs*

```
$ cat Cisco-IOS-XE-switch-deviation.yang

module Cisco-IOS-XE-switch-deviation {

  namespace "http://cisco.com/ns/yang/Cisco-IOS-XE-switch-deviation";

  prefix ios-sw-d;


  import Cisco-IOS-XE-native {

     prefix ios;

  }


  import Cisco-IOS-XE-policy {

     prefix ios-policy;

  }


  import Cisco-IOS-XE-switch {

     prefix ios-sw;

  }


------ The output is omitted for brevity ------


  deviation "/ios:native/ios:policy/ios-policy:class-map/" +

      "ios-policy:match/ios-policy:access-group/ios-policy:index" {

    deviate replace {

      type uint32 {

        range "1..2799";

      }

    }

  }

  deviation "/ios:native/ios:policy/ios-policy:policy-map/" +

      "ios-policy:class/ios-policy:action-list/ios-policy:action-param/" +

      "ios-policy:bandwidth-case/ios-policy:bandwidth/ios-policy:bits" {

    deviate replace {

      type uint32 {

    range "100..40000000";

      }

    }

  }

! The output is truncated for brevity
```

In Example 13-15, the **deviate** keyword is coupled with the action **replace**, which means that the original leaf node is replaced by the node contained in the **deviate replace** entry.

Note

For a more comprehensive list of actions, refer to RFC 6020.

YANG 1.1

YANG has evolved a lot since it was created and described in RFC 6020 in 2010. RFC 7950, "The YANG 1.1 Data Modeling Language," was released in 2016. This new RFC doesn't obsolete the original one; rather, it extends the capabilities of YANG in terms of supported features. In 2017, the IETF decided to change the structure of its YANG models to be compliant with the Network Management Datastore Architecture (NMDA); this change is defined in RFC 8342, which formally updates RFC 7950.

One of the most important changes is that YANG 1.1 modules have a stricter format and stricter character rules. It is important to check the consistency of a module with these new rules if you want to convert the module from YANG 1.0 to YANG 1.1.

You may be wondering how it is possible to know what version of YANG a particular module conforms to. Starting with Version 1.1, as specified in RFC 7950 Section 7.1.2, it's mandatory to include a YANG version statement in a header statement, as shown in Example 13-16.

Example 13-16 *YANG Version Within a Module*

```
$ cat openconfig-interfaces.yang

module openconfig-interfaces {


  yang-version "1";


  // namespace

  namespace "http://openconfig.net/yang/interfaces";


  prefix "ocif";

------ The output is truncated for brevity ------
```

Although the module in Example 13-16is developed using the YANG 1.0 language, it has the keyword **yang-version** in its header statements, which helps you (and applications using this module) understand which version rules should be applied. If the module corresponds to the newer version, the value of **yang-version** is **1.1**. If a YANG module doesn't contain this field, it automatically means that the module complies with the original YANG 1.0 version.

By now, you should be familiar with the concepts of the YANG language, the structure of a YANG module, and YANG components and data types. Understanding YANG isn't easy, and you might want to spend some time reviewing real YANG modules, such as those developed by Cisco (https://github.com/YangModels/yang) or OpenConfig (https://github.com/openconfig/public), to get a better grasp of the nuances of the languages.

The rest of this chapter is dedicated to explaining the differences between Cisco native and third-party YANG modules, as well as how to use popular Python tools to explore YANG modules.

Types of YANG Modules

Various YANG modules are available, some of them open-standard modules from the IETF and OpenConfig, and others vendor-specific modules from different vendors for different hardware and software platforms. The first step in using YANG modules for model-based automation is to locate and download the YANG modules that you plan on using.

The Home of YANG Modules

The examples illustrating the YANG language shown earlier in this chapter use real Cisco and third-party YANG modules. There are two ways to get such modules:

• Extract them from a network device.

• Download them from the Internet.

You can extract modules from a network device via protocol-specific NETCONF/RESTCONF/gRPC syntax, which you will learn in the upcoming chapters covering each protocol. Generally, this is the best way to go, as it means extracting all the YANG modules, including augmentations and deviations for the particular platform that you are working on. These extracted modules can be used later on to build NETCONF/RESTCONF/gRPC messages used in your scenarios for network automation.

Downloading modules from the Internet is easier than extracting modules from a network device because the only thing you need is Internet connectivity. Different vendors have different approaches to the distribution of their YANG modules. Some of them include this information in a package together with the network operating system software. Cisco makes its YANG modules open and available for everyone, and it publishes YANG modules directly on GitHub, as shown on the Figure 13-3.

GitHub is the largest platform for open-source projects and activities, and it provides capabilities to easily save and share information, such as program code, documentation, and other text documents. Also, it has a built-in version control system and capabilities for collaboration so that several people can work on the same project simultaneously.

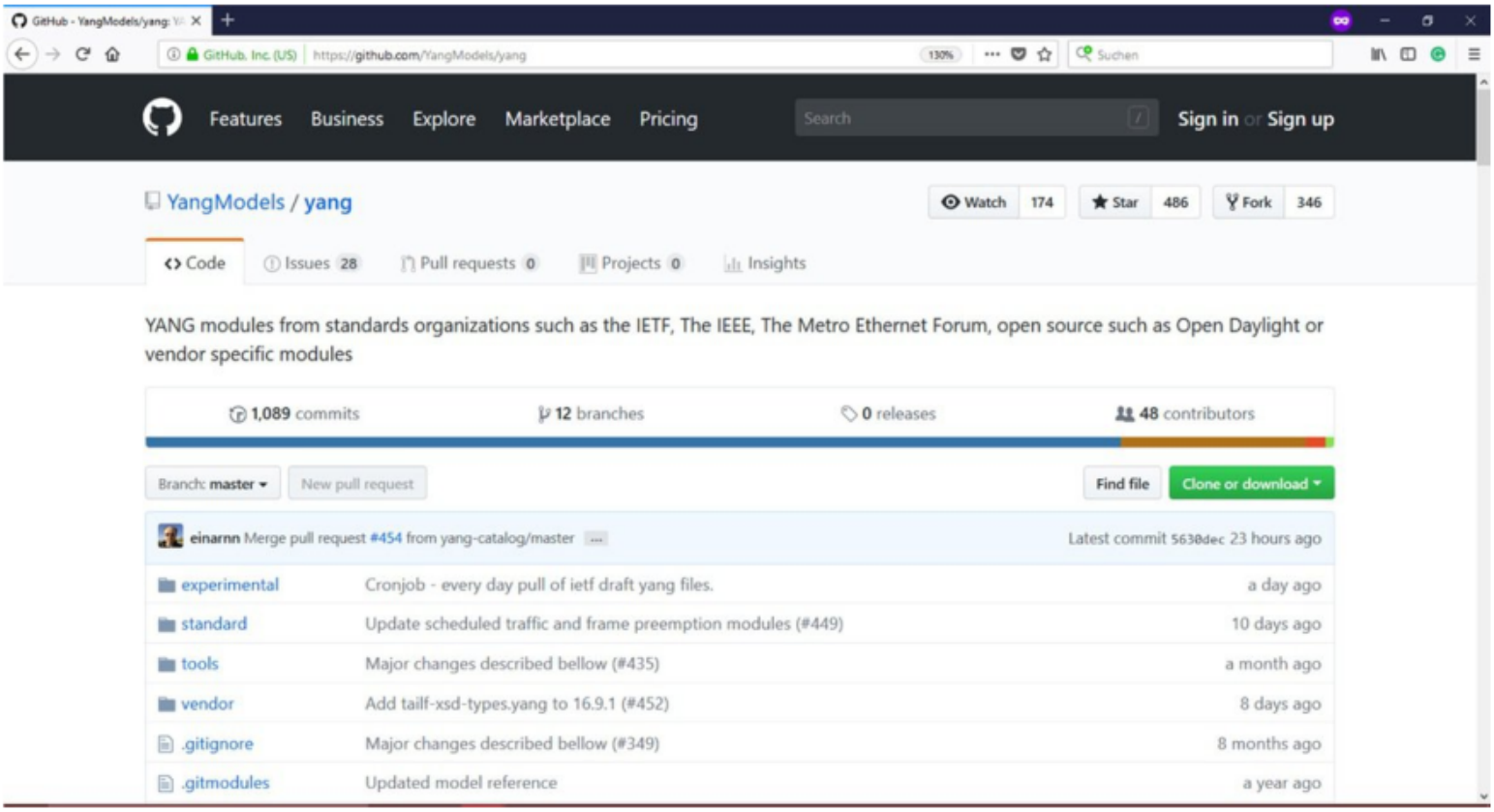

Figure 13-3 *GitHub Website with YANG Modules*

Figure 13-3 shows a page on the GitHub website that contains YANG modules from Cisco and other vendors (such as Juniper), as well as standardized IETF modules. It's possible to download files by simply clicking the button labeled Clone or Download, but there is a more convenient option: You can use the Linux **git** tool. Example 13-17 shows how to install the **git** tool on an rpm-based Linux distro such as CentOS.

Example 13-17 *Installing the **git** Tool on Linux CentOS*

```
$ sudo yum install -y git
```

The next step is to clone the repository of YANG modules locally to your PC so that you are can work with them using standard Linux tools, such as **cat**, **more**, and **vim**, as shown in Example 13-18.

Example 13-18 *Cloning YANG Modules from GitHub*

```
$ git clone https://github.com/YangModels/yang.git
Cloning into 'yang'...
remote: Enumerating objects: 2, done.
remote: Counting objects: 100% (2/2), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 21976 (delta 0), reused 1 (delta 0), pack-reused 21974
Receiving objects: 100% (21976/21976), 42.41 MiB | 2.95 MiB/s, done.
Resolving deltas: 100% (17077/17077), done.
Checking out files: 100% (20902/20902), done.
```

In the output shown in Example 13-18, you can see that all the data is cloned to the folder **yang**. Example 13-19 shows how to review what was cloned.

Example 13-19 *Verifying Downloaded YANG Modules*

```
$ ls -l yang/
total 24
drwxrwxr-x.  9 aaa aaa  127 Sep 22 17:53 experimental
-rw-rw-r--.  1 aaa aaa 9788 Sep 22 17:53 README.md
-rwxrwxr-x.  1 aaa aaa  502 Sep 22 17:53 setup.py
drwxrwxr-x.  7 aaa aaa   77 Sep 22 17:53 standard
-rwxrwxr-x.  1 aaa aaa  225 Sep 22 17:53 testall.sh
drwxrwxr-x. 15 aaa aaa 4096 Sep 22 17:53 tools
drwxrwxr-x.  6 aaa aaa   75 Sep 22 17:53 vendor
```

In Example 13-19, the hierarchy under the **yang** directory is exactly the same as in Figure 13-1. You can find more detailed explanation on the downloaded content in the README.md file by following common GitHub practice. Example 13-20 provides some excerpts from that file.

Example 13-20 *Information on Downloaded YANG Modules*

```
$ cat yang/README.md


------ The output is omitted for brevity ------


YANG

=====


This repository contains a collection of YANG modules:


  * IETF standards-track YANG modules

  * OpenDaylight open source YANG modules

  * IEEE experimental YANG modules

  * Broadband Forum standard YANG modules

  * Vendor-specific YANG modules

  * Open source YANG tools


------ The output is truncated for brevity ------
```

Native (Vendor-Specific) YANG Modules

So far this chapter has mostly looked at native Cisco YANG modules. This is the starting point for the development of YANG data models that all the vendors follow. The reason for that is obvious: It's much easier to develop a YANG data model around the existing configuration structure than to map third-party YANG models (such as IETF or OpenConfig models) to the configuration.

A downloaded YANG package contains all the available Cisco modules, distributed in folders based on the type of the network operating system, as shown in the Example 13-21.

Example 13-21 *YANG Modules by Operating System Type*

```
$ ls -l yang/vendor/cisco/

total 12

-rwxrwxr-x.  1 aaa aaa  564 Sep 22 17:53 check.sh

drwxrwxr-x.  2 aaa aaa   23 Sep 22 17:53 common

drwxrwxr-x. 15 aaa aaa 4096 Sep 22 17:53 nx

-rw-rw-r--.  1 aaa aaa  530 Sep 22 17:53 README.md

drwxrwxr-x. 11 aaa aaa  147 Sep 22 17:53 xe

drwxrwxr-x. 21 aaa aaa  248 Sep 22 17:53 xr
```

Cisco is famous for its excellent documentation, and it continues this tradition with YANG modules by putting notes in the README.md file, which provides plenty of useful information. Example 13-22 shows that the README.md file explains where you can find different YANG modules.

Example 13-22 *Built-in Guide for Cisco YANG Modules*

```
$ cat yang/vendor/cisco/README.md

This directory contains YANG models for Cisco platforms. There are several sub-

directories:


* **common** - models that have some level of support across all IOS-XR, NX-OS and IOS-

XE; there may be deviations either published by devices or available in OS-specific

directories

* **xr** - models that are specific to IOS-XR platforms

* **nx** - models that are specific to NX-OS platforms

* **xe** - models that are specific to IOS-XE platforms


Each subdirectory may have further OS/platform-specific information in a README file.
```

The folder for Cisco IOS XR YANG modules contains the README.md file shown in Example 13-23.

Example 13-23 *Built-in Guide for Cisco YANG Modules - continuation*

```
$ cat yang/vendor/cisco/xr/README.md

This directory contains OS/platform-specific YANG models for Cisco's IOS-XR platforms.


The directory is _currently_ organized by OS-version, with each sub-directory
```

```
containing the models for that version. The OS version number is presented as a single

number. Thus the YANG models for IOS-XR 5.3.0 will be in the subdirectory named "530".

Please note that this organization may change.


A README file may exist in the version subdirectories with any specific notes relating

to the models in the directory.


Further documentation on Cisco's IOS-XR YANG-based interfaces may be found at:


* [Cisco IOS XR Data Models Configuration Guide for the NCS 5500 Series

Router](http://www.cisco.com/c/en/us/td/docs/iosxr/ncs5500/DataModels/b-Datamodels-cg-

ncs5500.html)

* [Cisco Network Convergence System 5500 Series Configuration

Guides](http://www.cisco.com/c/en/us/support/routers/network-convergence-system-5500-

series/products-installation-and-configuration-guides-list.html)
```

As shown in Example 13-23, the README.md file provides clear explanation of the structure of the folders further down the directory hierarchy, which is arranged to have one subdirectory per Cisco IOS XR version. In each of these subdirectories, you can find all the supported YANG modules for the particular version of IOS XR. Example 13-24 shows several of the modules supported in Cisco IOS XR 6.5.1.

Example 13-24 *YANG Modules in Cisco IOS XR 6.5.1*

```
$ ls -l yang/vendor/cisco/xr/651/

total 14136

-rw-rw-r--. 1 aaa aaa  62027 Sep 22 17:53 capabilities-ncs1001.xml

-rw-rw-r--. 1 aaa aaa  66494 Sep 22 17:53 capabilities-ncs1k.xml

-rw-rw-r--. 1 aaa aaa  81766 Sep 22 17:53 capabilities-ncs5500.xml

-rw-rw-r--. 1 aaa aaa  70568 Sep 22 17:53 capabilities-ncs5k.xml

-rw-rw-r--. 1 aaa aaa  73592 Sep 22 17:53 capabilities-xrv9k.xml

-rw-rw-r--. 1 aaa aaa    591 Sep 22 17:53 capabilities-xrvr.xml

-rwxrwxr-x. 1 aaa aaa   2937 Sep 22 17:53 check-models.sh

-rw-rw-r--. 1 aaa aaa  17136 Sep 22 17:53 Cisco-IOS-XR-aaa-aaacore-cfg.yang

-rw-rw-r--. 1 aaa aaa   1871 Sep 22 17:53 Cisco-IOS-XR-aaa-diameter-base-mib-cfg.yang

-rw-rw-r--. 1 aaa aaa  12226 Sep 22 17:53 Cisco-IOS-XR-aaa-diameter-cfg.yang

-rw-rw-r--. 1 aaa aaa  28159 Sep 22 17:53 Cisco-IOS-XR-aaa-diameter-oper-sub1.yang

-rw-rw-r--. 1 aaa aaa   2969 Sep 22 17:53 Cisco-IOS-XR-aaa-diameter-oper.yang

-rw-rw-r--. 1 aaa aaa   6767 Sep 22 17:53 Cisco-IOS-XR-aaa-lib-cfg.yang

-rw-rw-r--. 1 aaa aaa   5621 Sep 22 17:53 Cisco-IOS-XR-aaa-lib-datatypes.yang

-rw-rw-r--. 1 aaa aaa   1040 Sep 22 17:53 Cisco-IOS-XR-aaa-li-cfg.yang


------ The output is truncated for brevity ------
```

The modules listed in Example 13-24 are the actual YANG modules that you will find on a device running IOS XR Version 6.5.1. If you are building network automation based on YANG, this folder will be your starting point for the YANG models for this specific platform. Also, as mentioned earlier, you can use these modules as examples to further your studies of YANG if you don't actually have a network device to experiment on.

Cisco follows the same approach for its other network operating systems: IOS XE and NX-OS. Example 13-25 shows the YANG modules for Cisco IOS XE 16.9.1.

Example 13-25 *YANG Modules in Cisco IOS XE 16.9.1*

```
$ ls -l yang/vendor/cisco/xe/1691/

total 6252

-rw-rw-r--. 1 aaa aaa  59225 Sep 22 17:53 asr1k-netconf-capability.xml

-rw-rw-r--. 1 aaa aaa  54180 Sep 22 17:53 asr920-netconf-capability.xml

-rw-rw-r--. 1 aaa aaa  52705 Sep 22 17:53 cat3k-netconf-capability.xml

-rw-rw-r--. 1 aaa aaa  54478 Sep 22 17:53 cat9300-netconf-capability.xml

-rw-rw-r--. 1 aaa aaa  54478 Sep 22 17:53 cat9400-netconf-capability.xml

-rw-rw-r--. 1 aaa aaa  54478 Sep 22 17:53 cat9500-netconf-capability.xml

-rw-rw-r--. 1 aaa aaa  60666 Sep 22 17:53 cbr-netconf-capability.xml

-rwxrwxr-x. 1 aaa aaa   3192 Sep 22 17:53 check-models.sh

-rw-rw-r--. 1 aaa aaa  14635 Sep 22 17:53 cisco-bridge-common.yang
```

```
-rw-rw-r--. 1 aaa aaa  35576 Sep 22 17:53 cisco-bridge-domain.yang
-rw-rw-r--. 1 aaa aaa   3134 Sep 22 17:53 cisco-ethernet.yang
-rw-rw-r--. 1 aaa aaa  21338 Sep 22 17:53 cisco-ia.yang
-rw-rw-r--. 1 aaa aaa  21110 Sep 22 17:53 Cisco-IOS-XE-aaa-oper.yang
-rw-rw-r--. 1 aaa aaa  98232 Sep 22 17:53 Cisco-IOS-XE-aaa.yang
-rw-rw-r--. 1 aaa aaa   2602 Sep 22 17:53 Cisco-IOS-XE-acl-oper.yang
-rw-rw-r--. 1 aaa aaa  39943 Sep 22 17:53 Cisco-IOS-XE-acl.yang
-rw-rw-r--. 1 aaa aaa   3682 Sep 22 17:53 Cisco-IOS-XE-arp-oper.yang


------ The output is truncated for brevity ------
```

Example 13-26 shows the YANG modules for Cisco NX-OS 9.2-1.

Example 13-26 *YANG Modules in Cisco IOS NX-OS 9.2-1*

```
$ ls -l yang/vendor/cisco/nx/9.2-1/
total 5468
-rwxrwxr-x. 1 aaa aaa    2153 Sep 22 17:53 check-models.sh
-rw-rw-r--. 1 aaa aaa    8665 Sep 22 17:53 cisco-nx-openconfig-acl-deviations.yang
-rw-rw-r--. 1 aaa aaa   11331 Sep 22 17:53 cisco-nx-openconfig-bgp-policy-
deviations.yang
-rw-rw-r--. 1 aaa aaa    1081 Sep 22 17:53 cisco-nx-openconfig-if-aggregate-
deviations.yang
-rw-rw-r--. 1 aaa aaa    1070 Sep 22 17:53 cisco-nx-openconfig-if-ethernet-
deviations.yang
-rw-rw-r--. 1 aaa aaa   25203 Sep 22 17:53 cisco-nx-openconfig-if-ip-deviations.yang
-rw-rw-r--. 1 aaa aaa    2461 Sep 22 17:53 cisco-nx-openconfig-if-ip-ext-
deviations.yang
-rw-rw-r--. 1 aaa aaa    4544 Sep 22 17:53 cisco-nx-openconfig-interfaces-
deviations.yang
-rw-rw-r--. 1 aaa aaa   90443 Sep 22 17:53 cisco-nx-openconfig-network-instance-
deviations.yang
-rw-rw-r--. 1 aaa aaa    2039 Sep 22 17:53 cisco-nx-openconfig-ospf-policy-
deviations.yang
-rw-rw-r--. 1 aaa aaa    3165 Sep 22 17:53 cisco-nx-openconfig-platform-deviations.yang
-rw-rw-r--. 1 aaa aaa     745 Sep 22 17:53 cisco-nx-openconfig-platform-linecard-
deviations.yang
-rw-rw-r--. 1 aaa aaa     886 Sep 22 17:53 cisco-nx-openconfig-platform-port-
deviations.yang


! The output is truncated for brevity
```

IETF YANG Modules

In the examples in this chapter, you might have noticed that almost all the modules for Cisco NX-OS include the keyword **openconfig**. These are vendor-independent YANG data models, and we examine them later in this chapter. Cisco NX-OS also has a vendor-specific data model, which is described by the module **Cisco-NX-OS-device.yang**.

A vendor-specific YANG module works only for a certain vendor and platform. For example, YANG modules for ASR 1000 routers running Cisco IOS XE don't work on Cisco ASR 9000 running Cisco IOS XR and vice versa. In a broader context, Cisco YANG modules don't work on Nokia SR 7750 running Nokia SR OS or Juniper MX routers running Junos OS. The same is true in the other direction: Juniper YANG modules work only with Junos OS platforms, and Nokia SR OS modules work only with Nokia routers. Hence, it is necessary to develop drivers for network automation based on YANG per vendor and per device.

To overcome the issues with vendor-specific YANG modules, YANG modules should be vendor neutral and implemented by all the vendors. The IETF has been leading the standardization of the different network technologies and protocols since the pre-Internet era. (The first IETF RFC dates to 1969.) The IETF also intends to create standardized (open) YANG modules, but as of this writing, not too many open modules are being integrated into network devices. Example 13-27 shows the IETF YANG modules in Cisco IOS XR.

Example 13-27 *IETF YANG Modules in Cisco IOS XR*

```
$ ls -l yang/vendor/cisco/xr/651/ | grep 'ietf'
-rw-rw-r--. 1 aaa aaa    921 Sep 22 17:53 cisco-xr-ietf-netconf-acm-deviations.yang
-rw-rw-r--. 1 aaa aaa   1106 Sep 22 17:53 cisco-xr-ietf-netconf-monitoring-
deviations.yang
```

```
-rw-rw-r--. 1 aaa aaa  16676 Sep 22 17:53 ietf-inet-types.yang
-rw-rw-r--. 1 aaa aaa  24295 Sep 22 17:53 ietf-interfaces.yang
-rw-rw-r--. 1 aaa aaa  12864 Sep 22 17:53 ietf-netconf-acm.yang
-rw-rw-r--. 1 aaa aaa  17518 Sep 22 17:53 ietf-netconf-monitoring.yang
-rw-rw-r--. 1 aaa aaa  26785 Sep 22 17:53 ietf-netconf.yang
-rw-rw-r--. 1 aaa aaa   4363 Sep 22 17:53 ietf-restconf-monitoring.yang
-rw-rw-r--. 1 aaa aaa   5320 Sep 22 17:53 ietf-syslog-types.yang
-rw-rw-r--. 1 aaa aaa   7035 Sep 22 17:53 ietf-yang-library.yang
-rw-rw-r--. 1 aaa aaa   4816 Sep 22 17:53 ietf-yang-smiv2.yang
-rw-rw-r--. 1 aaa aaa  18066 Sep 22 17:53 ietf-yang-types.yang
```

Compared to the number of native Cisco IOS XR modules, the number of IETF open modules is insignificant. Nevertheless, these vendor-neutral modules play an important role as they help unify different data types (such as interface types) across different vendors that implement and use these modules. One particular module, called ietf-netconf.yang, is of paramount importance. This module, which is available for each platform, describes the operation of the NETCONF protocol on that particular platform, including all available operations.

In this chapter, we have already discussed network device–level YANG modules, which are typically used for programming the network functions by using a network management system (NMS). Another type of YANG modules is used to control an NMS in terms of service provisioning by some external function (for example, a service orchestrator or OSS/BSS), and it requires a higher level of network abstraction. This level is called *service modeling*, and the YANG language is used to create a data model of the service. The IETF created YANG modules for such end-to-end services as well. For example, RFC 8299, "YANG Data Model for L3VPN Service Delivery," describes a YANG module for provisioning BGP IP VPN services. At a high level, this model describes the service characteristics, such as type of the service, access to external services out of the VPN (for example, on the Internet or in the cloud), and details of the IP addresses and circuit types for connecting the customer equipment. However, this RFC doesn't include all the information needed to configure the network devices, and it is used by an NMS as an input that should be enriched with further details and converted into a request using a device-specific YANG module.

OpenConfig YANG Modules

Although the IETF is a well-established standards development organization, its pace is sometimes quite slow compared with industry requirements. This is primarily due to the fact that all the different entities participating in the IETF must agree and come to a conclusion on each matter with which the IETF is involved. From a customer perspective, this process takes too long, especially in the context of YANG modules and associated use cases, which are developing at a significantly fast pace. Therefore, some industry heavy lifters, including Google, Comcast, Verizon, and Deutsche Telekom, got together and organized an informal working group called OpenConfig. The main intent of this initiative is to create vendor-neutral YANG modules and to push vendors to implement and support them. To keep it simple, a YANG module should be the same, regardless of vendor, so that an automation application can be developed once and then used with all network vendor equipment using the open YANG modules without much integration effort.

OpenConfig has a limited (though ever-growing) number of YANG modules. The initial focus for OpenConfig was to develop modules to help the customers deploy data centers very quickly. Now it also has support for a variety of MPLS modules, including modules for segment routing, VRF, routing protocols, route policies, and many other technologies and protocols involved in network device configuration. Nevertheless, OpenConfig still doesn't cover all possible network functions. For instance, it is not yet possible to create EVPN services by using OpenConfig YANG modules, though this might change in the future.

OpenConfig has its own page on GitHub from which you can download its YANG modules. The good news is that the process is the same as for downloading other modules from GitHub, as you can see in Example 13-28.

Example 13-28 *Downloading and Verifying OpenConfig YANG Modules*

```
$ git clone https://github.com/openconfig/public.git openconfig
Cloning into 'openconfig'...
remote: Enumerating objects: 46, done.
remote: Counting objects: 100% (46/46), done.
remote: Compressing objects: 100% (42/42), done.
remote: Total 1631 (delta 13), reused 16 (delta 4), pack-reused 1585
Receiving objects: 100% (1631/1631), 1.03 MiB | 1.34 MiB/s, done.
Resolving deltas: 100% (877/877), done.


$ ls -l openconfig/
total 16
drwxrwxr-x. 4 aaa aaa   198 Sep 23 11:24 doc
-rw-rw-r--. 1 aaa aaa 11358 Sep 23 11:24 LICENSE
-rw-rw-r--. 1 aaa aaa  1156 Sep 23 11:24 README.md
drwxrwxr-x. 3 aaa aaa    37 Sep 23 11:24 release
```

The development of OpenConfig YANG modules is primarily driven by customers, with vendors assisting with the implementation. This does not necessarily mean that the YANG modules officially published by the OpenConfig working group are all implemented by all vendors. Example 13-29 shows a comparison of the interface YANG modules officially published by OpenConfig and supported in Cisco IOS XR 6.5.1.

Example 13-29 *Official and Implemented Interface OpenConfig Modules in IOS XR*

```
$ ls -l yang/vendor/cisco/xr/651/ | grep 'openconfig-if-\|openconfig-interfaces'
-rw-rw-r--. 1 aaa aaa   1126 Sep 22 17:53 cisco-xr-openconfig-if-ethernet-
deviations.yang
```

```
-rw-rw-r--. 1 aaa aaa   5848 Sep 22 17:53 cisco-xr-openconfig-if-ip-deviations.yang

-rw-rw-r--. 1 aaa aaa   1042 Sep 22 17:53 cisco-xr-openconfig-interfaces-

deviations.yang

-rw-rw-r--. 1 aaa aaa   4212 Sep 22 17:53 openconfig-if-aggregate.yang

-rw-rw-r--. 1 aaa aaa   7733 Sep 22 17:53 openconfig-if-ethernet.yang

-rw-rw-r--. 1 aaa aaa  25769 Sep 22 17:53 openconfig-if-ip.yang

-rw-rw-r--. 1 aaa aaa  28225 Sep 22 17:53 openconfig-interfaces.yang

$ ls -l openconfig/release/models/interfaces/

total 108

-rw-rw-r--. 1 aaa aaa  5068 Sep 23 11:24 openconfig-if-aggregate.yang

-rw-rw-r--. 1 aaa aaa  3108 Sep 23 11:24 openconfig-if-ethernet-ext.yang

-rw-rw-r--. 1 aaa aaa 11217 Sep 23 11:24 openconfig-if-ethernet.yang

-rw-rw-r--. 1 aaa aaa  4055 Sep 23 11:24 openconfig-if-ip-ext.yang

-rw-rw-r--. 1 aaa aaa 36151 Sep 23 11:24 openconfig-if-ip.yang

-rw-rw-r--. 1 aaa aaa  1994 Sep 23 11:24 openconfig-if-poe.yang

-rw-rw-r--. 1 aaa aaa  2595 Sep 23 11:24 openconfig-if-tunnel.yang

-rw-rw-r--. 1 aaa aaa  2336 Sep 23 11:24 openconfig-if-types.yang

-rw-rw-r--. 1 aaa aaa 31942 Sep 23 11:24 openconfig-interfaces.yang
```

If you compare the output from the first command in Example 13-29 to the output from the second command, you see that Cisco IOS XR 6.5.1 implements roughly only half of the available OpenConfig modules for the interface. And almost all of the available modules have deviations tailoring the original scope to a particular platform. (Refer to the section "Deviations in YANG Modules," earlier in this chapter, for more information.) On the other hand, a true advantage of OpenConfig is that a lot of vendors support OpenConfig modules, so with proper testing, it's a really unified way of programming network functions. To emphasize this fact, some vendors, such as Arista, use OpenConfig YANG modules as their core data models and create native data models only to cover gaps where OpenConfig currently doesn't provide some needed functionality.

YANG Tools

Now that you are familiar with the YANG language and the different types of data models, the last important topic in this chapter is to YANG tools that enable you to view, edit, and use YANG models effectively.

Using pyang

Reading YANG modules directly isn't an easy task, especially if YANG modules have imported modules or just have a complex structure due to using groupings, derived data types, and so forth.

An important YANG tool is **pyang**. It's difficult to state just how important and useful this tool is with respect to network programmability using YANG. Among other tasks, **pyang** is able to do the following:

- Visualize YANG modules in human-readable output in CLI or HTML files
- Create JSON or XML schemas out of YANG modules
- Translates JSON to XML schemes
- Validate a YANG module, including imported and augmented modules and nodes

pyang is used to manage XML and JSON schemas. Chapters 10 and 11 provide information on XML and JSON This chapter shows the visualization of YANG modules using **pyang**.

pyang is part of the Python Software Foundation, and it's installed using the **pip** tool in Linux, as shown in Example 13-30. (**pip** installation is explained in Chapter 5, "Python Fundamentals.")

Example 13-30 *Installing the **pyang** Tool*

```
$ sudo pip install pyang
```

Alternatively, you can download the latest **pyang** software from GitHub and install it manually, but doing so is beyond the scope of this chapter.

When you have **pyang** installed, you can create a visualization of a YANG module. At the beginning of this chapter, the YANG module called **Cisco-IOS-XR-ifmgr-cfg.yang** was used a lot, and it is used in Example 13-31 as well.

Example 13-31 *Using **pyang** to Discover a YANG Module*

```
$ pyang -f tree -p yang/vendor/cisco/xr/651/ \

    yang/vendor/cisco/xr/651/Cisco-IOS-XR-ifmgr-cfg.yang


module: Cisco-IOS-XR-ifmgr-cfg

  +--rw global-interface-configuration

  |  +--rw link-status?   Link-status-enum

  +--rw interface-configurations

     +--rw interface-configuration* [active interface-name]

        +--rw dampening
```

```
|  +--rw args?                 enumeration
|  +--rw half-life?            uint32
|  +--rw reuse-threshold?      uint32
|  +--rw suppress-threshold?   uint32
|  +--rw suppress-time?        uint32
|  +--rw restart-penalty?      uint32
+--rw mtus
|  +--rw mtu* [owner]
|     +--rw owner    xr:Cisco-ios-xr-string
|     +--rw mtu      uint32
+--rw encapsulation
|  +--rw encapsulation?         string
|  +--rw capsulation-options?   uint32
+--rw shutdown?                      empty
+--rw interface-virtual?             empty
+--rw secondary-admin-state?         Secondary-admin-state-enum
+--rw interface-mode-non-physical?   Interface-mode-enum
+--rw bandwidth?                     uint32
+--rw link-status?                   empty
+--rw description?                   string
+--rw active                         Interface-active
+--rw interface-name                 xr:Interface-name
```

The syntax of the **pyang** command is as follows:

```
pyang -f {format} [-p {path}] {module}
```

There are different {*format*} options, each for a different representation or visualization of the information in the module. In Example 13-31, the {*format*} value is **tree**, which means the information in the YANG module is visualized in a tree-like hierarchy. The next parameter, **-p** {*path*}, informs **pyang** where to look for dependencies, such as imported modules. On the one hand, there are some predefined paths, so this attribute is optional, but typically you need to define it manually. Finally, {*module*} defines what particular module or modules should be reviewed and visualized.

It's important to note that you can simultaneously act on several modules by using **pyang**, and doing so makes sense if the modules augment each other. Example 13-32 illustrates such a case.

Example 13-32 *Using* ***pyang*** *to Review Augmented YANG Modules*

```
pyang -f tree -p yang/vendor/cisco/xr/651/ \
    yang/vendor/cisco/xr/651/Cisco-IOS-XR-ifmgr-cfg.yang \
    yang/vendor/cisco/xr/651/Cisco-IOS-XR-ipv4-io-cfg.yang


module: Cisco-IOS-XR-ifmgr-cfg
  +--rw global-interface-configuration
  |  +--rw link-status?   Link-status-enum
  +--rw interface-configurations
     +--rw interface-configuration* [active interface-name]
        +--rw dampening
        |  +--rw args?                 enumeration
        |  +--rw half-life?            uint32
        |  +--rw reuse-threshold?      uint32
        |  +--rw suppress-threshold?   uint32
        |  +--rw suppress-time?        uint32
        |  +--rw restart-penalty?      uint32
        +--rw mtus
        |  +--rw mtu* [owner]
        |     +--rw owner    xr:Cisco-ios-xr-string
        |     +--rw mtu      uint32
        +--rw encapsulation
        |  +--rw encapsulation?         string
        |  +--rw capsulation-options?   uint32
```

```
+--rw shutdown?                                empty
+--rw interface-virtual?                       empty
+--rw secondary-admin-state?                   Secondary-admin-state-enum
+--rw interface-mode-non-physical?             Interface-mode-enum
+--rw bandwidth?                               uint32
+--rw link-status?                             empty
+--rw description?                             string
+--rw active                                   Interface-active
+--rw interface-name                           xr:Interface-name
+--rw ipv4-io-cfg:ipv4-network
|  +--rw ipv4-io-cfg:bgp-pa
|  |  +--rw ipv4-io-cfg:input
|  |  |  +--rw ipv4-io-cfg:source-accounting?        boolean
|  |  |  +--rw ipv4-io-cfg:destination-accounting?   boolean
|  |  +--rw ipv4-io-cfg:output
|  |     +--rw ipv4-io-cfg:source-accounting?        boolean
|  |     +--rw ipv4-io-cfg:destination-accounting?   boolean
|  +--rw ipv4-io-cfg:verify
|  |  +--rw ipv4-io-cfg:reachable?      Ipv4-reachable
|  |  +--rw ipv4-io-cfg:self-ping?      Ipv4-self-ping
|  |  +--rw ipv4-io-cfg:default-ping?   Ipv4-default-ping
|  +--rw ipv4-io-cfg:bgp
|  |  +--rw ipv4-io-cfg:qppb
|  |  |  +--rw ipv4-io-cfg:input
|  |  |     +--rw ipv4-io-cfg:source?        Ipv4-interface-qppb
|  |  |     +--rw ipv4-io-cfg:destination?   Ipv4-interface-qppb
|  |  +--rw ipv4-io-cfg:flow-tag
|  |     +--rw ipv4-io-cfg:flow-tag-input
|  |        +--rw ipv4-io-cfg:source?        boolean
|  |        +--rw ipv4-io-cfg:destination?   boolean
|  +--rw ipv4-io-cfg:addresses
|  |  +--rw ipv4-io-cfg:secondaries
|  |  |  +--rw ipv4-io-cfg:secondary* [address]
|  |  |     +--rw ipv4-io-cfg:address      inet:ipv4-address-no-zone
|  |  |     +--rw ipv4-io-cfg:netmask      inet:ipv4-address-no-zone
|  |  |     +--rw ipv4-io-cfg:route-tag?   uint32
|  |  +--rw ipv4-io-cfg:primary!
|  |  |  +--rw ipv4-io-cfg:address      inet:ipv4-address-no-zone
|  |  |  +--rw ipv4-io-cfg:netmask      inet:ipv4-address-no-zone
|  |  |  +--rw ipv4-io-cfg:route-tag?   uint32
|  |  +--rw ipv4-io-cfg:unnumbered?    xr:Interface-name
|  |  +--rw ipv4-io-cfg:dhcp?          empty
|  +--rw ipv4-io-cfg:helper-addresses
|  |  +--rw ipv4-io-cfg:helper-address* [address vrf-name]
|  |     +--rw ipv4-io-cfg:address     inet:ipv4-address-no-zone
|  |     +--rw ipv4-io-cfg:vrf-name    xr:Cisco-ios-xr-string
|  +--rw ipv4-io-cfg:forwarding-enable?       empty
|  +--rw ipv4-io-cfg:icmp-mask-reply?         empty
|  +--rw ipv4-io-cfg:tcp-mss-adjust-enable?   empty
|  +--rw ipv4-io-cfg:ttl-propagate-disable?   empty
|  +--rw ipv4-io-cfg:point-to-point?          empty
|  +--rw ipv4-io-cfg:mtu?                     uint32
+--rw ipv4-io-cfg:ipv4-network-forwarding
   +--rw ipv4-io-cfg:directed-broadcast?   empty
```

```
          +--rw ipv4-io-cfg:unreachables?         empty
          +--rw ipv4-io-cfg:redirects?            Empty
```

In the output in Example 13-32, the augmenting YANG module has the prefix **ipv4-io-cfg** before its nodes, and this enables you to track how it's constructed. Another good thing is that the augmenting nodes are added exactly where needed, based on the **augment** keyword, as explained earlier in this chapter.

The output of **pyang** displays the structure of all the components in the YANG module (contained in the definition statements). The output also includes the data types associated with YANG node (**empty** or **unit32**, which are built-in data types, and **Interface-mode-enum**, which is a derived data type), as well as the allowed actions for the node: A node labeled **ro** is a read-only node and is a state note representing one piece of operational data, and a node labeled **rw** is a read/write node and represents configuration data. Cisco IOS XR has dedicated models for operational data only and other models for configuration data only. Example 13-33 shows the operational data YANG model that corresponds to the same entities that are configurable through the module in Example 13-32.

Example 13-33 *Using **pyang** to Reviewa single YANG Module*

```
$ pyang -f tree -p yang/vendor/cisco/xr/651/ \
    yang/vendor/cisco/xr/651/Cisco-IOS-XR-ifmgr-oper.yang


module: Cisco-IOS-XR-ifmgr-oper
  +--ro interface-dampening
  |  +--ro interfaces
  |  |  +--ro interface* [interface-name]
  |  |     +--ro if-dampening
  |  |     |  +--ro interface-dampening
  |  |     |  |  +--ro penalty?                 uint32
  |  |     |  |  +--ro is-suppressed-enabled?   boolean
  |  |     |  |  +--ro seconds-remaining?       uint32
  |  |     |  |  +--ro flaps?                   uint32
  |  |     |  |  +--ro state?                   Im-state-enum
  |  |     |  +--ro state-transition-count?       uint32
  |  |     |  +--ro last-state-transition-time?   uint32
  |  |     |  +--ro is-dampening-enabled?         boolean
  |  |     |  +--ro half-life?                    uint32
  |  |     |  +--ro reuse-threshold?              uint32
  |  |     |  +--ro suppress-threshold?           uint32
  |  |     |  +--ro maximum-suppress-time?        uint32
  |  |     |  +--ro restart-penalty?              uint32


------ The output is truncated for brevity ------
```

If you want to save the representation of a YANG module or if you prefer graphical representation rather than the CLI, you can generate the HTML representation of the YANG module by using **pyang** as well. Example 13-34 shows how to generate such a file.

Example 13-34 *Using **pyang** to save the output of the jstree representation of a YANG Module*

```
$ pyang -f jstree -o test.html -p openconfig/release/models/ \
     openconfig/release/models/interfaces/openconfig-interfaces.yang \
     openconfig/release/models/interfaces/openconfig-if-*
```

Compared to the earlier examples of **pyang**, the syntax in Example 13-34 is extended with optional attribute [**-o** {*output*}], which points to the file where the output should be saved. In addition, the format of the data representation set by the **-f** key is different and is now called **jstree**, where the first two letters, **js**, stand for JavaScript. You can view the HTML file that is output in any browser, as shown in Figure 13-4.

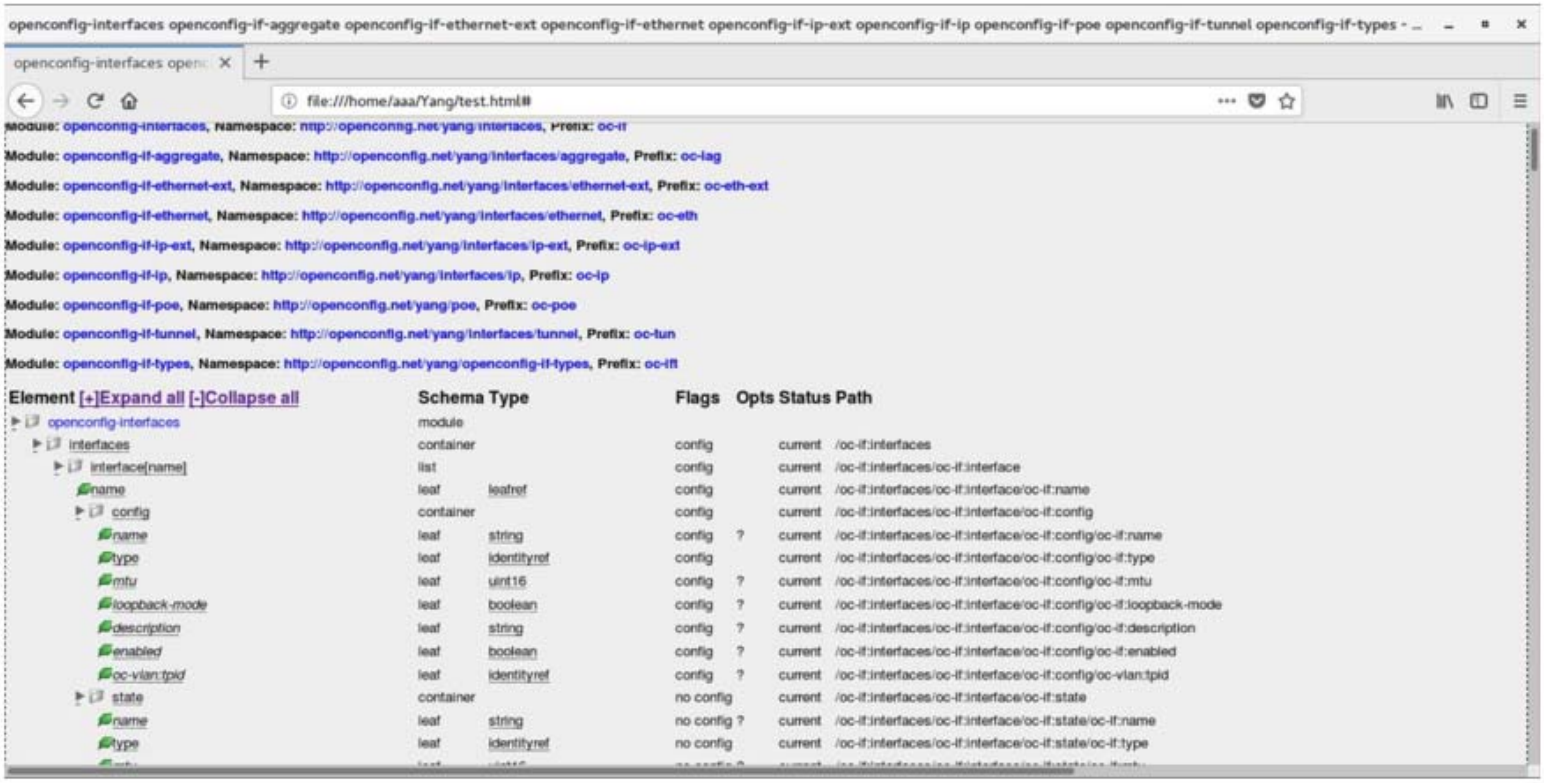

Figure 13-4 *HTML Representation of the YANG Modules Generated by YANG*

Example 13-34 generates a visualization of the OpenConfig YANG data model for the configuration of the interfaces, including all augmented YANG modules. There is some additional information, including the types of the YANG nodes, whether they are configurable, and an absolute path to the YANG node.

Using pyangbind

Chapter 5 and Chapter 6, "Python Applications," discuss the Python language and its ecosystem. **pyang** has a plug-in called **pyangbind** that helps translate YANG modules directly into Python classes so you can use Python to manage network elements through NETCONF or gNMI with YANG data modules.

In later chapters in this book, you will see some practical examples on how you can use **pyangbind**. For now, it's important to install it and gain a basic understanding of how it works. But before installing **pyangbind**, you need to make sure that all prerequisites are installed, as shown in Example 13-35 for Linux CentOS.

Example 13-35 *Installing Mandatory Packages for **pyangbind** in CentOS 7*

```
$ sudo yum -y install gcc gcc-c++ kernel-devel

$ sudo yum -y install python-devel libxslt-devel libffi-devel openssl-devel
```

After the mandatory packages are installed, you can install **pyangbuild**, as shown in Example 13-36.

Example 13-36 *Installing **pyangbind***

```
$ sudo pip install pyangbind

Collecting pyangbind

  Using cached https://files.pythonhosted.org/packages/2e/20/7b3f2de320d120e845bce141148a23522fccb089b

76ae0c76b5d5623d515/pyangbind-0.8.1.tar.gz

Requirement already satisfied: pyang in /usr/lib/python2.7/site-packages (from

pyangbind) (1.7.5)

Collecting bitarray (from pyangbind)

  Using cached https://files.pythonhosted.org/packages/e2/1e/b93636ae36d08d0ee3aec40b08731cc97217c69db

9422c0afef6ee32ebd2/bitarray-0.8.3.tar.gz

Requirement already satisfied: lxml in /usr/lib64/python2.7/site-packages (from

pyangbind) (4.2.4)

Collecting regex (from pyangbind)

  Using cached https://files.pythonhosted.org/packages/2a/0a/944977367c8a6cfcfa6fcb8ac6b1f0f9a667c1f34

194091c766b5d7c44d7/regex-2018.08.29.tar.gz

Requirement already satisfied: six in /usr/lib/python2.7/site-packages (from pyangbind)

(1.14.0)

Requirement already satisfied: enum34 in /usr/lib/python2.7/site-packages (from

pyangbind) (1.0.4)

Installing collected packages: bitarray, regex, pyangbind
```

```
  Running setup.py install for bitarray ... done
  Running setup.py install for regex ... done
  Running setup.py install for pyangbind ... done
Successfully installed bitarray-0.8.3 pyangbind-0.8.1 regex-2018.8.29
```

Note

If you haven't installed the required mandatory packages first, the setup of **pyangbind** won't be successful.

---

pyangbind is considered to be a plug-in for **pyang**, and they are used together. As explained earlier, **pyangbid** is used to convert YANG modules into Python classes so that they can be used in network applications. Example 13-37 shows a short bash script to run **pyangbind**.

Example 13-37 *Preparing to Run* ***pyang***

```
$ cat test_pyangbind.sh
#!/bin/bash

PYBINDPLUGIN=`/usr/bin/env python -c 'import pyangbind; import os; print "%s/plugin" %
os.path.dirname(pyangbind.__file__)'`
pyang --plugindir $PYBINDPLUGIN -p yang/vendor/cisco/xr/651 -f pybind -o
test_interfaces.py yang/vendor/cisco/xr/651/Cisco-IOS-XR-ifmgr-cfg.yang

echo "Bindings successfully generated!"
```

Example 13-37 uses a Bash script to ease the structure of the **pyang** command. As you can see, the path to the directory containing **pyangbind** alongside the necessary attributes is very long and complex. Offloading it to a dedicated variable allows you to decouple the path to the plug-in from the **pybind** command itself. Then comes the **pyang** command, which you should already be familiar with. What is new here is the format used in the **pyang** command (**-f pybind**) and the path to the plug-in directory configured as **--plugindir** {*link*}. The remaining part of the **pyang** command, including the path to the folder and the actual modules, isn't new. The last part of the command sends a message to the terminal in the event that the classes are successfully generated. In Example 13-38, you can see the result of executing the **pyangbind** script.

Example 13-38 *Executing the* ***pyangbind*** *Script*

```
$ ./test_pyangbind.sh
Bindings successfully generated!
```

As you can see in Example 13-38, the CLI output indicates that the script was successfully executed. You now know you can check the result in the file, which was provided using the **-o** { *file*} parameter in the initial command (refer to Example 13-37). Example 13-39 shows the output of that file.

Example 13-39 *Result of the* ***pyangbind*** *Class Mapping*

```
$ cat test_interfaces.py
# -*- coding: utf-8 -*-
from operator import attrgetter
from pyangbind.lib.yangtypes import RestrictedPrecisionDecimalType
from pyangbind.lib.yangtypes import RestrictedClassType
from pyangbind.lib.yangtypes import TypedListType
from pyangbind.lib.yangtypes import YANGBool
from pyangbind.lib.yangtypes import YANGListType
from pyangbind.lib.yangtypes import YANGDynClass
from pyangbind.lib.yangtypes import ReferenceType
from pyangbind.lib.base import PybindBase
from collections import OrderedDict
from decimal import Decimal
from bitarray import bitarray
import six

# PY3 support of some PY2 keywords (needs improved)
if six.PY3:
  import builtins as __builtin__
  long = int
elif six.PY2:
  import __builtin__
```

```
class
yc_global_interface_configuration_Cisco_IOS_XR_ifmgr_cfg__global_interface_configuratio
n(PybindBase):
  """
  This class was auto-generated by the PythonClass plugin for PYANG
  from YANG module Cisco-IOS-XR-ifmgr-cfg - based on the path /global-interface-
configuration. Each member element of
  the container is represented as a class variable - with a specific
  YANG type.

  YANG Description: Global scoped configuration for interfaces
  """
  __slots__ = ('_path_helper', '_extmethods', '__link_status',)

  _yang_name = 'global-interface-configuration'

  _pybind_generated_by = 'container'

  def __init__(self, *args, **kwargs):
    self._path_helper = False
    self._extmethods = False
    self.__link_status = YANGDynClass(base=RestrictedClassType(base_type=six.text_type,
                                  restriction_type="dict_key",
                      restriction_arg={u'default': {u'value': 0}, u'software-
interfaces': {u'value': 2}, u'disable': {u'value': 1}},), default=six.text_type("default"),
is_leaf=True, yang_name="link-status", parent=self, path_helper=self._path_helper,
extmethods=self._extmethods, register_paths=True, namespace='http://cisco.com/ns/yang
/Cisco-IOS-XR-ifmgr-cfg', defining_module='Cisco-IOS-XR-ifmgr-cfg', yang_type='Link-
status-enum', is_config=True)
    load = kwargs.pop("load", None)
    if args:
      if len(args) > 1:
        raise TypeError("cannot create a YANG container with >1 argument")
      all_attr = True
      for e in self._pyangbind_elements:
        if not hasattr(args[0], e):
          all_attr = False
          break
      if not all_attr:
        raise ValueError("Supplied object did not have the correct attributes")
      for e in self._pyangbind_elements:
        nobj = getattr(args[0], e)
        if nobj._changed() is False:
          continue
        setmethod = getattr(self, "_set_%s" % e)
        if load is None:
          setmethod(getattr(args[0], e))
        else:
          setmethod(getattr(args[0], e), load=load)
  def _path(self):
    if hasattr(self, "_parent"):
      return self._parent._path()+[self._yang_name]
    else:
      return [u'global-interface-configuration']
```

```
  def _get_link_status(self):


! The output is truncated for brevity
```

Although the output in Example 13-39 is extensive, it shows you how YANG can be automatically translated to Python classes that you can use in your Python code.

Using pyang to Create JTOX Drivers

There is one more useful scenario where **pyang** can be helpful. The NETCONF protocol uses XML encoding, but for humans, it is easier to work with JSON. **pyang** has a specific output format, which is a JTOX (JSON-to-XML) driver. As the name implies, JTOX allows you to convert a JSON file to XML; this is possible if JSON is created using the same YANG module as the JTOX driver.

To implement a JTOX solution, you need to identify the YANG module or modules that will be used to create a JTOX driver. You can determine which module you need by verifying the attachment point for augmentation in YANG modules, as described earlier in this chapter (refer to Example 13-12). When you find necessary modules, you list all of them as input for **pyang**, as shown in Example 13-40. (Because the syntax of this command is explained earlier in this chapter, it isn't explained again here.)

Example 13-40 *Creating a JTOX Driver Out of Several YANG Modules*

```
$ pyang -f jtox -o cisco_if.jtox -p 612/ 612/Cisco-IOS-XR-ifmgr-cfg.yang 612/Cisco-IOS-
XR-ipv4-io-cfg.yang 612/Cisco-IOS-XR-ipv6-ma-cfg.yang 612/Cisco-IOS-XR-drivers-media-
eth-cfg.yang
612/Cisco-IOS-XR-drivers-media-eth-cfg.yang:13: warning: imported module Cisco-IOS-XR-
types not used
```

The output in Example 13-40 contains a warning that some of the imported modules aren't used. This doesn't have any impact on the resulting JTOX driver, but it indicates that YANG modules probably import something that isn't referenced in this output. The reason could be that the content of the YANG module was changed, but the imported modules weren't reviewed.

Example 13-41 shows the output of the JTOX driver from Example 13-40 that is composed from several YANG modules.

Example 13-41 *JTOX Driver Created from Several YANG Modules*

```
$ cat cisco_if.jtox
{"tree": {"Cisco-IOS-XR-ifmgr-cfg:interface-configurations": ["container", {"interface-
configuration": ["list", {"dampening": ["container", {"args": ["leaf", "enumeration"],
"suppress-threshold": ["leaf", "uint32"], "half-life": ["leaf", "uint32"], "suppress-
time": ["leaf", "uint32"], "reuse-threshold": ["leaf", "uint32"], "restart-penalty":
["leaf", "uint32"]}], "description": ["leaf", "string"], "secondary-admin-state":
["leaf", "enumeration"], "interface-virtual": ["leaf", "empty"], "Cisco-IOS-XR-drivers-
media-eth-cfg:ethernet": ["container", {"inter-packet-gap": ["leaf", "enumeration"],
"signal-fail-bit-error-rate": ["container", {"signal-remote-fault": ["leaf", "empty"],
"signal-fail-report-disable": ["leaf", "empty"], "signal-fail-threshold": ["leaf",
"uint32"]}], "duplex": ["leaf", "enumeration"], "speed": ["leaf", "enumeration"],
"loopback": ["leaf", "enumeration"], "forward-error-correction": ["leaf",
"enumeration"], "priority-flow-control": ["leaf", "enumeration"], "auto-negotiation":
["leaf", "enumeration"], "signal-degrade-bit-error-rate": ["container", {"signal-
degrade-report": ["leaf", "empty"], "signal-degrade-threshold": ["leaf", "uint32"]}],
"carrier-delay": ["container", {"carrier-delay-up": ["leaf", "uint32"], "carrier-delay-
down": ["leaf", "uint32"]}], "flow-control": ["leaf", "enumeration"]}], "Cisco-IOS-XR-
ipv6-ma-cfg:ipv6-network": ["container", {"bgp-flow-tag-policy-table": ["container",
{"bgp-flow-tag-policy": ["container", {"source": ["leaf", "boolean"], "destination":
["leaf", "boolean"]}]}]}], "unnumbered": ["leaf", "string"], "addresses": ["container",
{"auto-configuration": ["container", {"enable": ["leaf", "empty"]}], "eui64-addresses":
["container", {"eui64-address": ["list", {"route-tag": ["leaf", "uint32"], "prefix-
length": ["leaf", "uint32"], "zone": ["leaf", "string"], "address": ["leaf", ["union",
["string", "string"]]]}, [["Cisco-IOS-XR-ipv6-ma-cfg", "address"]]]}], "regular-
addresses": ["container", {"regular-address": ["list", {"route-tag": ["leaf",
"uint32"], "prefix-length": ["leaf", "uint32"], "zone": ["leaf", "string"], "address":
["leaf", ["union", ["string", "string"]]]}, [["Cisco-IOS-XR-ipv6-ma-cfg",
"address"]]]}], "link-local-address": ["container", {"route-tag": ["leaf", "uint32"],
"zone": ["leaf", "string"], "address": ["leaf", ["union", ["string", "string"]]]}]}],
"tcp-mss-adjust-enable": ["leaf", "empty"], "verify": ["container", {"default-ping":
["leaf", "enumeration"], "self-ping": ["leaf", "enumeration"], "reachable": ["leaf",
```

```
"enumeration"]]], "bgp-policy-accountings": ["container", {"bgp-policy-accounting":
["list", {"direction": ["leaf", "string"], "destination-accounting": ["leaf",
"boolean"], "source-accounting": ["leaf", "boolean"]}, [["Cisco-IOS-XR-ipv6-ma-cfg",
"direction"]]]}], "mac-address-filters": ["container", {"mac-address-filter": ["list",
{"multicast-address": ["leaf", "string"]}, [["Cisco-IOS-XR-ipv6-ma-cfg", "multicast-
address"]]]}], "ttl-propagate-disable": ["leaf", "empty"], "unreachables": ["leaf",
"empty"], "mtu": ["leaf", "uint32"], "bgp-qos-policy-propagation": ["container",
{"source": ["leaf", "enumeration"],
!
! The output is truncated for brevity
```

As you can see, the JTOX driver in Example 13-41 is much longer than the one in Example 13-42. Despite the difference in length, the structure hasn't changed much. The only significant new point is highlighted in Example 13-42; when the augmentation should be done, it is called using the syntax { *module-name:top_container* }, in the same way that the parent module is called. You should consider this when constructing a JSON object.

Example 13-42 shows a JSON object for a fully functional Ethernet interface built using augmenting YANG modules.

Example 13-42 *A JSON Object with an Interface Configuration on Cisco IOS XR*

```
$ cat cisco_if.json
{
    "Cisco-IOS-XR-ifmgr-cfg:interface-configurations": {
        "interface-configuration": [
            {
                "active": "act",
                "interface-name": "GigabitEthernet0/0/0/0",
                "mtus": {
                        "mtu": [
                            {
                                "mtu": 1514,
                                "owner": "GigabitEthernet"
                            }
                        ]
                },
                "Cisco-IOS-XR-drivers-media-eth-cfg:ethernet": {
                    "carrier-delay": {
                        "carrier-delay-down": 0,
                        "carrier-delay-up": 0
                    }
                },
                "Cisco-IOS-XR-ipv4-io-cfg:ipv4-network": {
                    "addresses": {
                        "primary": {
                            "address": "10.11.33.33",
                            "netmask": "255.255.255.0"
                        }
                    ]
                },
                "Cisco-IOS-XR-ipv6-ma-cfg:ipv6-network": {
                    "addresses": {
                        "regular-addresses": {
                            "regular-address": [
                                {
                                    "address": "fc00::10:11:33:33",
                                    "prefix-length": 112,
                                    "zone": 0
                                }
                            ]
```

```
                        }
                    }
                }
            }
        ]
    }
}
```

Following the syntax of the JTOX driver, the JSON object is composed using proper nodes and calling augmenting modules, where necessary. As you can see, the configuration of IPv4 and IPv6 addresses requires additional YANG modules. The same is true for Ethernet parameters, and this construction is becoming common in network programming based on YANG.

Example 13-43 shows the process of transforming a JSON object to XML and the resulting XML file.

Example 13-43 *Transforming JSON to XML by Using JTOX*

```
$ json2xml -t config -o test_book.xml cisco_if.jtox cisco_if.json
$ cat test_book.xml
<?xml version='1.0' encoding='utf-8'?>
<nc:config xmlns:drivers-media-eth-cfg="http://cisco.com/ns/yang/Cisco-IOS-XR-drivers-
media-eth-cfg" xmlns:ifmgr-cfg="http://cisco.com/ns/yang/Cisco-IOS-XR-ifmgr-cfg"
xmlns:ipv4-io-cfg="http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-io-cfg" xmlns:ipv6-ma-
cfg="http://cisco.com/ns/yang/Cisco-IOS-XR-ipv6-ma-cfg"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:inet="urn:ietf:params:xml:ns:yang:ietf-inet-types" xmlns:ip-iarm-
datatypes="http://cisco.com/ns/yang/Cisco-IOS-XR-ip-iarm-datatypes"
xmlns:xr="http://cisco.com/ns/yang/cisco-xr-types">
  <ifmgr-cfg:interface-configurations>
    <ifmgr-cfg:interface-configuration>
      <ifmgr-cfg:active>act</ifmgr-cfg:active>
      <ifmgr-cfg:interface-name>GigabitEthernet0/0/0/0</ifmgr-cfg:interface-name>
      <ifmgr-cfg:mtus>
        <ifmgr-cfg:mtu>
          <ifmgr-cfg:owner>GigabitEthernet</ifmgr-cfg:owner>
          <ifmgr-cfg:mtu>1514</ifmgr-cfg:mtu>
        </ifmgr-cfg:mtu>
      </ifmgr-cfg:mtus>
      <drivers-media-eth-cfg:ethernet>
        <drivers-media-eth-cfg:carrier-delay>
          <drivers-media-eth-cfg:carrier-delay-up>0</drivers-media-eth-cfg:carrier-delay-up>
          <drivers-media-eth-cfg:carrier-delay-down>0</drivers-media-eth-cfg:carrier-delay-down>
       </drivers-media-eth-cfg:carrier-delay>
      </drivers-media-eth-cfg:ethernet>
      <ipv4-io-cfg:ipv4-network>
        <ipv4-io-cfg:addresses>
          <ipv4-io-cfg:primary>
            <ipv4-io-cfg:netmask>255.255.255.0</ipv4-io-cfg:netmask>
            <ipv4-io-cfg:address>10.11.33.33</ipv4-io-cfg:address>
          </ipv4-io-cfg:primary>
        </ipv4-io-cfg:addresses>
      </ipv4-io-cfg:ipv4-network>
      <ipv6-ma-cfg:ipv6-network>
        <ipv6-ma-cfg:addresses>
          <ipv6-ma-cfg:regular-addresses>
            <ipv6-ma-cfg:regular-address>
              <ipv6-ma-cfg:address>fc00::10:11:33:33</ipv6-ma-cfg:address>
              <ipv6-ma-cfg:prefix-length>112</ipv6-ma-cfg:prefix-length>
              <ipv6-ma-cfg:zone>0</ipv6-ma-cfg:zone>
```

```
            </ipv6-ma-cfg:regular-address>
          </ipv6-ma-cfg:regular-addresses>
        </ipv6-ma-cfg:addresses>
      </ipv6-ma-cfg:ipv6-network>
    </ifmgr-cfg:interface-configuration>
  </ifmgr-cfg:interface-configurations>
</nc:config>
```

The most crucial point that you should pay attention to in the output of Example 13-43 is how different XML namespaces are mapped to the parts of the configuration file. Compared to Example 13-28, four different namespaces are actively used in the *<config>* context:

- **xmlns:ifmgr-cfg="http://cisco.com/ns/yang/Cisco-IOS-XR-ifmgr-cfg"**
- **xmlns:ipv4-io-cfg="http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-io-cfg"**
- **xmlns:ipv6-ma-cfg="http://cisco.com/ns/yang/Cisco-IOS-XR-ipv6-ma-cfg"**
- **xmlns:drivers-media-eth-cfg="http://cisco.com/ns/yang/Cisco-IOS-XR-drivers-media-eth-cfg"**

All the relevant information for defining these XML namespaces exists in the JTOX driver, as shown in Example 13-44.

Example 13-44 *Multiple YANG Module Definitions in a JTOX Driver*

```
$ cat cisco_if.jtox
! The output is truncated for brevity
"modules": {"Cisco-IOS-XR-ipv4-io-cfg":
["ipv4-io-cfg", "http://cisco.com/ns/yang/Cisco-IOS-XR-ipv4-io-cfg"], "Cisco-IOS-XR-drivers-media-eth-
cfg": ["drivers-media-eth-cfg", "http://cisco.com/ns/yang/Cisco-IOS-XR-drivers-media-
eth-cfg"], "Cisco-IOS-XR-ifmgr-cfg": ["ifmgr-cfg", "http://cisco.com/ns/yang/Cisco-IOS-
XR-ifmgr-cfg"], "Cisco-IOS-XR-ip-iarm-datatypes": ["ip-iarm-datatypes",
"http://cisco.com/ns/yang/Cisco-IOS-XR-ip-iarm-datatypes"], "ietf-inet-
types": ["inet", "urn:ietf:params:xml:ns:yang:ietf-inet-types"], "Cisco-IOS-XR-ipv6-ma-
cfg": ["ipv6-ma-cfg", "http://cisco.com/ns/yang/Cisco-IOS-XR-ipv6-ma-cfg"], "Cisco-IOS-XR-types":
["xr", "http://cisco.com/ns/yang/cisco-xr-types"]}, "annotations": {}}
```

In Example 13-44, the names of YANG modules called in the XML file that was created out of the JSON/JTOX transformation are highlighted.

Summary

This chapter covers the following topics:

- YANG is a data modeling language for creating data models that can be used anywhere, including for network management and programmability.
- The YANG data model has a tree structure.
- There are four types of nodes in a YANG tree: leaf, leaf-list, container, and list nodes.
- A grouping is a bundle of different nodes that can be reused in different contexts.
- A YANG module can contain data or can import information from other YANG modules.
- The YANG language allows you to extend a parent YANG data model with additional nodes; this is called augmentation.
- Sometimes it is necessary to tailor the scope of the initial YANG data models or to set some limits for just some cases or network functions; this is called deviation.
- Native (vendor-specific) YANG modules follow the native system's configuration structure.
- Third-party YANG modules provide a vendor-neutral view of the network devices data model across all the vendors.
- **pyang** validates and converts original YANG modules in human-readable output or in other types used in network automation applications.
- **pyangbind** makes it possible to use YANG modules directly from Python.
- JTOX is used to convert a network function configuration represented in JSON into XML format that is consumable by NETCONF.
- **json2xml** can be used for JSON-to-XML transformation based on the JTOX driver.

Part VII: Protocols

Chapter 14. NETCONF and RESTCONF

Protocols build on the functions provided by the transport, encoding, and modeling layers of the network programmability stack to provide a complete framework for automating the provisioning, operation, and maintenance of networks. This chapter discusses the NETCONF and RESTCONF protocols. NETCONF was developed first and provides an RPC-based API. RESTCONF, which is a RESTful protocol, was developed later and provides a subset of NETCONF functionality.

NETCONF

As you will see in this section, NETCONF relies heavily on SSH, XML, and YANG. This section also covers working with NETCONF using Python. Therefore, to make the most of this section, you need to be sure you've went through all the relevant chapters that cover these subjects.

NETCONF Overview

In 2003, the IETF assembled the NETCONF Working Group (later renamed the Network Configuration Working Group), tasked with developing a protocol to address the shortcomings of existing practices and protocols for configuration management, such as SNMP. The working group's solution to these shortcomings was the NETCONF protocol. The background work preceding the design phase of NETCONF is documented in RFC 3535, "Overview of the 2002 IAB Network Management Workshop." The design goals from that work include the following:

- Make a distinction between configuration and state data
- Create multiple configuration data stores (candidate, running, and startup)
- Record configuration change transactions
- Ensure configuration testing and validation support
- Enable selective data retrieval with filtering
- Enable streaming and playback of event notifications
- Create an extensible procedure call mechanism

The NETCONF Working Group's activities (in chronological order) are listed at https://datatracker.ietf.org/wg/netconf/history/.

NETCONF, which stands for Network Configuration Protocol, can be used to configure a device, retrieve configuration or state data from a device, or issue exec mode commands, as long as the device is running a NETCONF server, or in other words, the device exposes a NETCONF API. Therefore, NETCONF is actually a misnomer because the protocol's function is not limited to *configuring* devices. NETCONF is formally defined in a number of RFCs. RFC 6241 covers the core protocol, which obsoletes the original RFC 4741. A number of other RFCs cover a variety of enhancements and updates to the protocol, as well as possible variations and/or implementations of the protocol. For example, RFC 6242 covers NETCONF over SSH, and RFCs 5717 and 6243 cover extensions to the base protocol. RFC 6244 explicitly covers the application of NETCONF to model-based programmability using YANG models. The following is a list of the RFCs for NETCONF (not including obsolete ones, as of this writing):

- RFC 5277, "NETCONF Event Notifications"
- RFC 5381, "Experience of Implementing NETCONF over SOAP (Informational)"
- RFC 5717, "Partial Lock Remote Procedure Call (RPC) for NETCONF"
- RFC 6022, "YANG Module for NETCONF Monitoring"
- RFC 6241, "Network Configuration Protocol (NETCONF)"
- RFC 6242, "Using the NETCONF Protocol over SSH"
- RFC 6243, "With-defaults Capability for NETCONF"
- RFC 6244, "An Architecture for Network Management Using NETCONF and YANG"
- RFC 7589, "Using the NETCONF Protocol over Transport Layer Security (TLS) with Mutual X.509 Authentication"
- RFC 7803, "Changing the Registration Policy for the NETCONF Capability URNs Registry (Best Current Practice)"
- RFC 8341, "Network Configuration Access Control Model"
- RFC 8526, "NETCONF Extensions to Support the Network Management Datastore Architecture"

NETCONF is a client/server session-based protocol. This means that a client initiates a connection to the server (the network device, in this case), such as an SSH connection to a particular TCP port that the server is listening on. When the server accepts the connection, both peers exchange protocol messages. Then the connection is torn down by one of the peers, either gracefully because the message exchange is complete or ungracefully because something went wrong.

After the client/server session is established, the client and server exchange hello messages, which provide information on which version of NETCONF is supported on each endpoint, as well other device capabilities. Capabilities describe which components of the NETCONF protocol, as well as which data models, the device supports. Support for some components is mandatory, and it is optional for others. Hello messages are not periodic; that is, after the initial hello message exchange, no further hello messages are exchanged.

Once the hello message exchange is completed, one or more *remote procedure call message* (rpc for short) are sent by the client. RPCs provide a programmatic method for a client to *call* (execute) a *procedure* (a piece of code) on a different device (which is why these calls are labeled *remote*). Each of the rpc messages specifies an *operation* for the server to carry out. An operation could be, for example, retrieving the running configuration of the device or editing the configuration.

The server executes the operation specified in the rpc message (or not) and responds with a *remote procedure call reply message* (rpc-reply for short) back to the client. The contents of the rpc-reply message depend on the operation requested by the client, the parameters included in the message, and whether the operation execution was successful.

The client/server session is terminated by one of the peers if one of many conditions becomes true. In a best-case scenario, the client sends an rpc message to the server, explicitly requesting that the connection be gracefully terminated. The server terminates the session, and the transport connection is torn down. On the other side of the spectrum, the transport connection may be unexpectedly lost due to a problem, and the server unilaterally kills the session.

Each of the concepts described in this section is covered in a lot of detail throughout the chapter.

Note

NETCONF messages from the client encoded in XML are characterised by the root element <rpc> and the NETCONF responses back from the server, also encoded in XML, are characterised by the root element <rpc-reply>. For better readability throughout the chapter, client messages will be referred to *rpc messages* and the server responses will be referred to as *rpc-reply messages*.

Figure 14-1 illustrates the high-level operation of NETCONF.

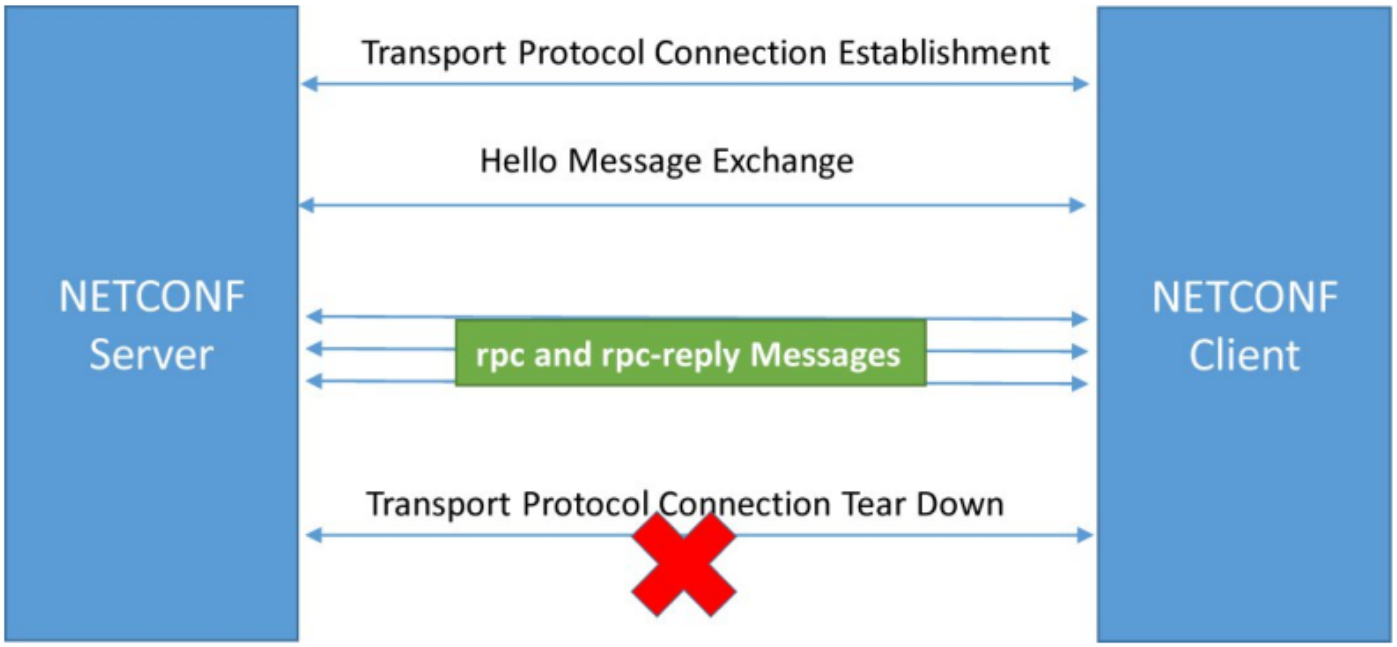

Figure 14-1 *High-Level Operation of NETCONF*

NETCONF Architecture

Like any other protocol, NETCONF is composed of several small functional components. Each of these components resides in one of the layers of a conceptual four-layer model, where each layer represents a group of similar functions. The functions contained in each layer of the model are independent of the functions in the other layers. Figure 14-2 illustrates this model.

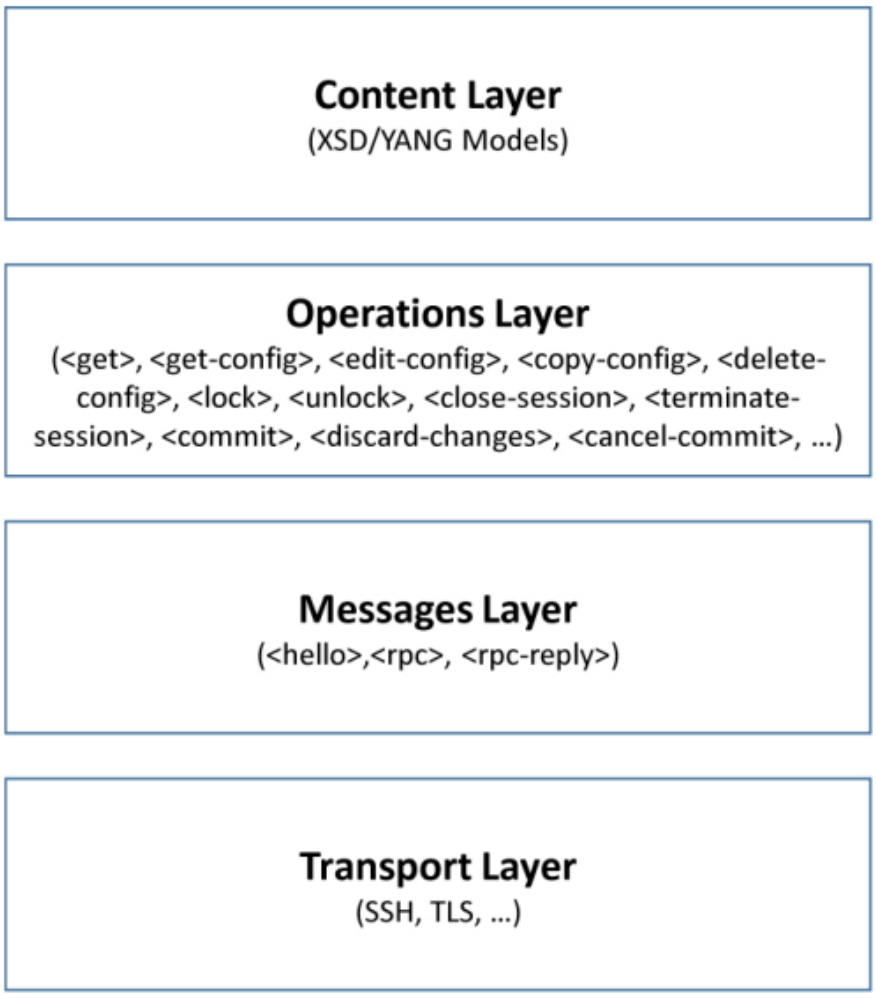

Figure 14-2 *The Four-Layer Model Encompassing All NETCONF Functions*

The transport layer at the bottom represents the functions performed by the transport protocol used by NETCONF to establish a secure, persistent connection between the client on one side and the device (server) on the other side. Although the majority of NETCONF implementations today use SSH for transport, any transport protocol can be used, as long as this protocol satisfies a number of criteria. However, it is a mandatory requirement for any NETCONF implementation to support SSH. The section "The NETCONF Transport Layer," later in this chapter, discusses these criteria as well as the specific implementation of NETCONF over SSH.

On top of the transport layer is the messages layer. This layer encompassed the RPC paradigm used by NETCONF. Both rpc and rpc-reply messages are covered in detail later in this chapter, in section "The NETCONF Messages Layer."

The operations layer describes all the actions, or operations, that the client can execute remotely on the device via an rpc message. An operation may retrieve configuration or state data from the device, or it might edit the running or startup configuration on the device. There are operations defined in the base NETCONF protocol, and there are extended operations that require special capabilities, which are advertised in the hello message. Client operations are discussed in detail in section "The NETCONF Operations Layer."

The top layer of the model, the content layer, describes the actual content that goes into the messages. Any NETCONF message is a well-formed XML document, and XML documents are composed of namespaces, elements, tags, entities, and attributes. But how do the values of these document components relate to device configurations or operational data? When NETCONF first came out, the data in NETCONF messages was unstructured. A few years ago, NETCONF implementations started using document type definitions (DTDs) or XML Schema Definition to define the structure of the data inside the message. Today, the data is modeled almost exclusively in YANG. NETCONF and YANG are covered in section "The NETCONF Content Layer."

The NETCONF Transport Layer

The first layer of the NETCONF protocol stack is the transport layer, which encompasses the functions necessary to establish the client/server connection required by NETCONF to operate.

NETCONF Transport Protocol Requirements

The NETCONF standard does not mandate implementing NETCONF over SSH *only*. (See Section 2 in both RFC 6241 and RFC 6244.) NETCONF can operate over any transport protocol that satisfies a minimum number of requirements. The standard does mandate, however, that any NETCONF implementation *must*, at a minimum, support NETCONF over SSH, and it may optionally support other transport protocols as well.

For NETCONF to use any transport protocol, the transport protocol must be capable of the following characteristics and functions:

• **Connection oriented:** NETCONF is a client/server protocol, which means it requires a transport protocol that is capable of establishing a persistent connection that is long-lived for the duration of a session, involving an arbitrary number of rpc and rpc-reply messages.

• **Reliable:** Because the transport protocol is connection oriented, NETCONF requires reliability and, hence, NETCONF runs over TCP-based transport protocols only. UDP is not a supported option. For example, NETCONF over TLS is supported, but NETCONF over DTLS is not.

• **Sequenced data delivery:** Being connection oriented also implies that the transport protocol must have mechanisms to reorder any out-of-sequence frames received at either end of the connection. Think of TCP sequence and acknowledgement numbers.

• **Authentication:** NETCONF fully delegates the authentication of the client by the server, and vice versa, to the transport protocol. This process is entirely transparent to NETCONF. However, once the transport protocol authenticates a user, it passes the username of that user to the upper layers of NETCONF, and it is then referred to as the *NETCONF username*. The NETCONF stack (not the transport protocol) then takes care of the authorization of that authenticated user identified by its NETCONF username.

• **Data integrity and confidentiality:** The transport protocol must maintain data integrity against malicious or non-intentional corruption. It must also implement the encryption mechanisms necessary to maintain data confidentiality.

NETCONF over SSH

As mentioned earlier, any NETCONF implementation *must* support SSH as a transport protocol. RFC 6242 (which obsoletes RFC 4742) is dedicated to NETCONF over SSH. Chapter 9, "SSH," discusses three SSH protocols: the transport, authentication, and connection protocols (also covered in RFC 4252, 4253 and 4254 respectively.) These three protocols work together to initiate the SSH session over which NETCONF operates and then to authenticate the client and server, maintain the session, and finally tear it down.

First, the transport protocol establishes an SSH transport connection from client to server. The client and server then exchange keys that are used for data integrity and encryption. The authentication protocol then kicks in, and the **ssh-userauth** service authenticates the user (client) to the server. Finally, the SSH connection protocol brings up the SSH session via the **ssh-connection** service.

After the SSH session is established, the NETCONF protocol stack invokes NETCONF as an SSH subsystem. The IANA-assigned TCP port for NETCONF is 830. However, this port is often configurable. A NETCONF over SSH session may be initiated in a number of ways. The simplest way is to use a terminal command such as:

```
[NetDev@server1~]$ ssh {username}@{device_address} -p {netconf_port}
```

Or alternatively:

```
[NetDev@server1~]$ ssh {username}@{device_address} -s {netconf_subsystem}
```

where **-p** is used to indicate the NETCONF port configured on the system (830 if left to the default), and **-s** is used to indicate the SSH subsystem name, which is usually **netconf** on most platforms. Which method to use depends on the platform you are trying to access. Some platforms use the first method, others use the second method, and some platforms support both methods. With the first method, the client initiates a TCP session to the port specified in the command and the NETCONF server, running on the device, listens to incoming TCP connections on that same port. In the second method, the client initiates a TCP session to the default SSH port (port 22), and then the SSH server on the device hands over the connection to the NETCONF subsystem running on the device. The second method may be more handy in situations where a firewall is sitting between the client and server, and only connections to the default SSH port are permitted.

The examples in this chapter use one of the IOS XE sandboxes provided by Cisco DevNet. The NETCONF API on the router is reachable through SSH to ios-xe-mgmt-latest.cisco.com on port 10000. The sandbox is always on (that is, no reservation is required) and is available to access and use for free. With this router, you use the username developer and the password C1sco12345. This router uses the first method, so you need to use the command **ssh -p 10000 admin@ ios-xe-mgmt-latest.cisco.com** to establish a NETCONF session to the router. You can also follow the examples in this chapter by using any other method you find convenient, such as your own lab routers or the Cisco Modeling Labs (CML).

Note

This router is up and running at the time of this writing, but its status may change at any time. You are strongly encouraged to check out the

numerous sandboxes that Cisco provides (currently for free) at https://devnetsandbox.cisco.com/RM/Topology.

The NETCONF Messages Layer

The three types of messages defined in NETCONF are hello, rpc, and rpc-reply messages. Hello messages are the first NETCONF messages exchanged between a client and a server when a NETCONF session is opened. Rpc messages are then sent by a client to request specific operations on a server, and the server responds with rpc-reply messages back to the client; the content of these messages depends on the operation in the rpc message and the result of that operation on the server.

Any valid NETCONF message must be a well-formed XML document encoded in UTF-8. A server returns an error message to the client if the client sends a NETCONF message that is either not well-formed or not encoded in UTF-8.

A NETCONF message may optionally start with an XML declaration, but the first mandatory line in the message is the root element, which must be one of three values—<hello>, <rpc>, or <rpc-reply>—depending on the type of message.

Regardless of the type of message or the version of NETCONF that is being used, the message elements are *always* defined in the namespace urn:ietf:params:xml:ns:netconf:base:1.0. Child elements down the hierarchy may possibly be defined in other namespaces. These namespaces point to the data models in which the elements are defined.

Following the root element, the child elements included in each message depend on the type of the message, the purpose of the message, and the data models referenced by the message elements. Regardless of the message type, all messages end in the character sequence]]>]]>.

Note

If what you just read in the previous few paragraphs does not sound very familiar, this may be a good time to go back and review Chapter 10, "XML"

Note

NETCONF is currently at Version 1.1, but at the time of this writing, Version 1.0 is still in wide use. Devices that support NETCONF Version 1.1 have the capability urn:ietf:params:netconf:base:1.1 advertised in their hello messages (as you will see in Example 14-1). NETCONF Version 1.1 uses a message framing mechanism called the *chunked framing mechanism*, in which a message is split into chunks. Each chunk includes the chunk size and chunk data, and the message is terminated using the character sequence \n##\n. To keep the examples simple, this chapter terminates messages using the character sequence]]>]]>, as this does not affect the core protocol functionality discussed in the chapter. Version 1.1 of the protocol primarily provides more capabilities beyond those provided by Version 1.0, but the core protocol remains the same. Apart from the framing mechanism, this chapter covers Version 1.1, unless otherwise stated.

Hello Messages

Once an SSH session is established by the transport layer, the client and server exchange hello messages. The messages are not exchanged in any particular order: The client and server may even send their hellos simultaneously. Example 14-1 shows a sample hello message from a Cisco router running IOS XE. The router sends this XML output to the terminal as soon as the password is entered correctly and the SSH session is established. (The router supports more than 200 capabilities, and the message is truncated here for brevity.)

Example 14-1 *Hello Message from the Server to the Client*

```
[NetDev@localhost ~]$ ssh developer@ios-xe-mgmt-latest.cisco.com -p 10000

developer@ios-xe-mgmt-latest.cisco.com's 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:xpath:1.0</capability>

        <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>

        <capability>urn:ietf:params:netconf:capability:validate:1.1</capability>

        <capability>urn:ietf:params:netconf:capability:rollback-on-

error:1.0</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=explicit&amp;also-supported=report-all-tagged</capability>

        <capability>urn:ietf:params:netconf:capability:yang-library:1.0?revision=2016-

06-21&amp;module-set-id=730825758336af65af9606c071685c05</capability>


--------- output omitted for brevity ---------


    </capabilities>

    <session-id>746</session-id>
```

```
</hello>]]>]]>
```

The hello message from the client can be as simple as the one shown in Example 14-2.

Example 14-2 *Hello Message from the Client to the Server*

```
<?xml version="1.0" encoding="UTF-8"?>

  <hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">

    <capabilities>

      <capability>urn:ietf:params:netconf:base:1.0</capability>

    </capabilities>

  </hello>]]>]]>
```

The hello message in Example 14-2 is what you would typically copy and paste into the terminal in order to complete the NETCONF session establishment.

Capabilities are discussed in detail later in this chapter. For now, notice the element hierarchy in the hello message. The message is composed of the root element <hello>, defined in the mandatory namespace urn:ietf:params:xml:ns:netconf:base:1.0, using the attribute xmlns. Under the root element is the <capabilities> element, which in turn contains a list of sibling <capability> elements, each containing one capability supported by the device.

The hello message from the server contains a mandatory <session-id> element, which contains the session ID of that particular session. The session ID is an important parameter that comes into play when a NETCONF session attempts to kill another session, maybe because that first session has a lock on a configuration datastore, and an administrator needs to terminate that session because it is hogging the device.

The client hello message, on the other hand, does not have a <session-id> element.

rpc Messages

Example 14-3 shows a NETCONF rpc message from a client requesting the interface admin state for interface GigabitEthernet1 on an IOS XE router.

Example 14-3 *An rpc Message for Retrieving the Admin State of Interface GigabitEthernet1*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="105">

    <get>

        <filter type="subtree">

            <interfaces xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-interfaces-oper">

                <interface>

                    <name>GigabitEthernet1</name>

                    <admin-status />

                </interface>

            </interfaces>

        </filter>

    </get>

</rpc>]]>]]>
```

In Example 14-3, the first element in the message is the root element <rpc>, which is defined in the mandatory namespace urn:ietf:params:xml:ns:netconf:base:1.0 using the attribute xmlns. Following the root <rpc> element down in the hierarchy is the operation. In this example, the operation is get, and hence the element is <get>.

The child elements that go under the operation element (<get> in this case) depend on two main factors. The first factor is what operation is requested in the rpc message. Some operations have mandatory elements. For example, the <get-config> operation has a mandatory element named <source>, which is used to specify the source datastore from which the configuration will be retrieved. Other elements are optional. Operations and their corresponding XML elements are discussed in detail in section "The Operations Layer," later in this chapter.

Of particular importance is the filter element. If the filter element is not included, the <get> and <get-config> operations retrieve all information, unfiltered, whether state and configuration data (in the case of the get operation) or configuration data only (in the case of the <get-config> operation). Filtering is also discussed in detail later in this chapter.

The second factor that determines which child elements go under the operation element is the data model that you wish to reference. In this example, the interfaces element and all its child elements are defined in the namespace http://cisco.com/ns/yang/Cisco-IOS-XE-interfaces-oper, which represents the YANG data model referenced by the specific hierarchy in the message (interfaces → interface → name | admin-status). The relationship between NETCONF and YANG data models is discussed in detail in section "The Content Layer," later in this chapter.

In addition to the XML namespace declared in the <rpc> root element, another mandatory attribute is message-id. The client uses an arbitrary number in the first rpc message it sends over a session, and it increments this value with each new message it sends to the server. The server saves the value of this attribute and attaches it in the corresponding rpc-reply message so that the client can identify which rpc-reply belongs to each rpc message. This enables the message pipelining capability in NETCONF: The client can send more than one rpc message before it receives any rpc-reply messages. However, the server should process the rpc messages in the order in which they were received.

In addition to the two mandatory attributes, a client can add any number of additional attributes to the <rpc> element in an rpc message, and those attributes will be mirrored back, unchanged, in the rpc-reply message.

rpc-reply Messages

Example 14-4 shows the rpc-reply message received from the IOS XE router in response to the rpc message in Example 14-3.

Example 14-4 *An rpc-reply Message for Retrieving the Admin State of Interface GigabitEthernet1*

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="105">
```

```
    <data>
        <interfaces xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-interfaces-oper">
            <interface>
                <name>GigabitEthernet1</name>
                <admin-status>if-state-up</admin-status>
            </interface>
        </interfaces>
    </data>
</rpc-reply>]]>]]>
```

In Example 14-4, the first element in the message is the root element, <rpc-reply>, defined in the mandatory namespace urn:ietf:params:xml:ns:netconf:base:1.0 using the attribute xmlns. The mandatory message-id attribute is also in the root element, mirroring the value received in the RPC message.

What follows the root <rpc-reply> element down in the hierarchy depends on the operation and the result of running that operation on the server. Generally speaking, three cases describe the possible elements in an rpc-reply message:

• The rpc message might contain an operation that results in data being retrieved and sent back in the rpc-reply message, such as the <get> and <get-config> operations. If the operation was executed successfully on the server, the element following the root <rpc-reply> element in the message is the <data> element, which encapsulates the retrieved data from the server. This is the case in Example 14-4.

• The rpc message might contain an operation that does not result in the retrieval of any data. For example, the <edit-config> operation requests that the configuration on the device be changed, and the <commit> operation requests that the configuration in the candidate datastore be committed to the running configuration. If the operation is successful, the element following the root <rpc-reply> element is an empty <ok> element, expressed as <ok/>.

• Some error might occur, maybe because the rpc message is not a well-formed XML document, or the operation in the rpc message is not executed on the server for some reason. In this case, the element following the root <rpc-reply> element is an <rpc-error> element.

Example 14-5 shows the rpc-reply message that is received when an error is injected into the rpc message in Example 14-3 when the value of the operation is changed from <get> to <get-INVALID-VALUE>.

Example 14-5 *An rpc-reply Message Indicating a Syntax Error in One of the Elements in the RPC Message*

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="105">
 <rpc-error>
  <error-type>protocol</error-type>
  <error-tag>unknown-element</error-tag>
  <error-severity>error</error-severity>
  <error-path>
      /rpc
  </error-path>
  <error-info>
    <bad-element>get-INVALID-VALUE</bad-element>
  </error-info>
  </rpc-error>
</rpc-reply>]]>]]>
```

The rpc-error element has a number of child elements that elaborate on the error condition:

• **error-type**: A string value that indicates the layer at which the error was found and has one of four values:

• **transport**

• **rpc**

• **protocol**

• **application**

• **error-tag**: A string value that describes the error. See Appendix A of RFC 6241 for a list of allowed values.

• **error-severity**: The error severity; the only allowed value is error.

• **error-app-tag**: A string value that describes the data model–specific or implementation–specific error condition.

• **error-path**: The absolute XPath to the element causing the error.

• **error-message**: A human-readable string that describes the error condition.

• **error-info**: Protocol-specific or data model–specific detailed error content.

As you can see from the sample error element in Example 14-5, not all child elements in this list are mandatory elements.

The NETCONF Operations Layer

Before we discuss the different operations defined by the NETCONF protocol, two general concepts need to be elaborated first: configuration data versus state data, and datastores.

The NETCONF protocol makes a clear distinction between configuration data and state data. Configuration data is, obviously, the configuration stored on the device. That configuration is what the device operator enters on the device in an attempt to bring the device to a desired operational state. However, the fact that specific configuration has been applied to the device does not necessarily mean that the operational state will follow. There are instances in which the state of a device does not follow the configuration on that device. Administratively enabling an interface does not necessarily mean that the interface will be in the up state. State data can either be the operational state of the different components of the device, such as the state of an interface (whether it is up or down), or it can be the statistics collected from the device, such as the interface incoming and outgoing packet count.

NETCONF supports different datastores. The concept of a datastore should be familiar to network engineers who have to save the running configuration to the startup configuration after changing the former to make sure the device will boot up from the updated configuration on the next reboot. In this case, the running configuration is one datastore, and the startup configuration is another. Some devices do not have a startup configuration datastore, and some devices support a candidate configuration datastore on which configuration changes are applied before being committed to the running configuration. What datastore is available for you to work on depends on the device. The only mandatory datastore on any device is the running configuration datastore. For example, IOS XR–based devices only have running and candidate configuration datastores and no startup configuration datastore.

An rpc message sent by a client to a server is a request for the server to execute a specific operation on the server and return an appropriate response to the client in an rpc-reply message. The operations layer defines a set of operations that are encapsulated in rpc messages that cover the scope of all allowed operations. The following sections loosely classify the operations into groups, based on what each operation is intended to accomplish.

Retrieving Data: <get> and <get-config>

The <get> and <get-config> operations are used to retrieve data from the server. The <get> operation retrieves all configuration and state data, and the <get-config> operation retrieves the configuration data from only a specific datastore.

The <source> element is a mandatory element for the <get-config> operation that indicates the configuration datastore from which the configuration will be retrieved. It can be <running>, <startup>, or <candidate>, depending on which datastores exist on the device and depending on the NETCONF capabilities supported by the device. Example 14-6 shows an rpc message to retrieve *all* the running configuration on an IOS XE router.

Example 14-6 *An rpc Message to Retrieve the Running Configuration on an IOS XE Router*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>]]>]]>]]>]]>
```

Example 4-7 shows partial output from the rpc message in Example 14-6.

Example 14-7 *The rpc-reply Message Containing the Running Configuration on the IOS XE Router*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <version>16.11</version>
            <boot-start-marker />
            <boot-end-marker />
            <banner>
                <motd>
                    <banner>^C</banner>
                </motd>
            </banner>
            <memory>
                <free>
                    <low-watermark>
                        <processor>80557</processor>
                    </low-watermark>
                </free>
            </memory>
            <call-home>
                <contact-email-addr xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-call-
home">sch-smart-licensing@cisco.com</contact-email-addr>
                <profile xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-call-home">
                    <profile-name>CiscoTAC-1</profile-name>
                    <active>true</active>
```

```
--------- output truncated for brevity ---------
```

If the <get> or <get-config> operation is successful, the data retrieved is encapsulated in a <data> element, as you can see in Example 14-7. If there is an error processing the rpc message, an rpc-reply message with an appropriate <rpc-error> element is returned.

In very rare cases you will want to retrieve the full configuration from a datastore or all state data on a device. Unless the full configuration is required, you will use the <filter> element to specify which parts of the data you need to retrieve. Filters exist in two flavors: subtree and XPath filters.

Subtree Filters

Subtree filters indicate the XML element subtrees to include in the output. To understand this, consider the output in Example 14-7. The root element of the message is the <rpc-reply> element, followed by the <data> element, as expected. The first child element under the <data> element is the <native> element, which is followed by several elements that are child elements to <native> and sibling elements to each other. Where the <native> element comes from is discussed in detail in the next section, "The Content Layer." For now, just accept it as one layer of hierarchy under which all configuration data exists.

Now say that you need the rpc-reply message to include only the IOS XE version, which is included in the output under the **<**version> element and is one layer of hierarchy under the <native> element. Then the <filter> element in the rpc message simply includes the <version> element, indicating that this element is to be included in the output data. Example 14-8 shows how the rpc message looks in this case.

Example 14-8 *The rpc Message to Retrieve the IOS XE Version Number*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <get-config>
    <source>
        <running/>
    </source>
    <filter type="subtree">
      <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
           <version></version>
      </native>
    </filter>
  </get-config>
</rpc>]]>]]>
```

Notice the type attribute, which should be equal to subtree or omitted altogether since this is the default value of the attribute. Example 14-9 shows the resulting rpc-reply message.

Example 14-9 *The rpc-reply Message, Including Only the IOS XE Version Number*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <version>16.11</version>
        </native>
    </data>
</rpc-reply>]]>]]>
```

This simple example illustrates the inclusive natures of subtree filters; that is, what you include in a filter appears in the output. Not including a <filter> element in the rpc message yields *everything*, and an empty <filter> element yields *nothing*. Example 14-8 is just meant to get you started, and subtree filters can be a little more sophisticated than that. Subtree filters can be based on five different components:

- Namespace selection
- Containment nodes
- Selection nodes
- Content match nodes
- Attribute match expressions

Namespace selection simply means pointing to the data model that defines the element hierarchy in the filter by using the **xmlns** attribute. (You don't need to worry about this just yet. The next section breaks down the relationship between YANG models, namespaces, and element hierarchies.)

Any element in an XML document that has child elements is called a *containment node*. When a subtree filter specifies a containment node in its hierarchy without any further criteria, the matching node in the data model (referenced by the namespace) is included in the output, along with all its child elements. In Example 14-10, the containment node <interface> is used to list the configuration for all interfaces on the router.

Example 14-10 *Using a Containment Node in a Subtree Filter*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <get-config>
    <source>
```

```
        <running/>
    </source>
    <filter>
      <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
        <interface></interface>
      </native>
    </filter>
  </get-config>
</rpc>]]>]]>
```

The output under the <data> element includes the element <interface> and all its child elements and all *their* child elements, all the way to the leaf nodes, as shown in Example 14-11.

Example 14-11 *The Result of Using a Containment Node in a Subtree Filter*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <GigabitEthernet>
                    <name>1</name>
                    <description>MANAGEMENT INTERFACE - DON'T TOUCH ME</description>
                    <ip>
                        <address>
                            <primary>
                                <address>10.10.20.48</address>
                                <mask>255.255.255.0</mask>
                            </primary>
                        </address>
                    </ip>
                    <mop>
                        <enabled>false</enabled>
                        <sysid>false</sysid>
                    </mop>
                    <negotiation xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ethernet">
                        <auto>true</auto>
                    </negotiation>
                </GigabitEthernet>
                <GigabitEthernet>
                    <name>2</name>
                    <description>Network Interface</description>
                    <ip>
                        <address>
                            <primary>
                                <address>10.10.10.10</address>
                                <mask>255.255.255.0</mask>
                            </primary>
                        </address>
                    </ip>
                    <mop>
                        <enabled>false</enabled>
                        <sysid>false</sysid>
                    </mop>
                    <negotiation xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ethernet">
                        <auto>true</auto>
```

```
                </negotiation>
            </GigabitEthernet>
            <GigabitEthernet>
                <name>3</name>
                <description>Network Interface</description>
                <shutdown />
                <mop>
                    <enabled>false</enabled>
                    <sysid>false</sysid>
                </mop>
                <negotiation xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ethernet">
                    <auto>true</auto>
                </negotiation>
            </GigabitEthernet>
        </interface>
    </native>
</data>
</rpc-reply>]]>]]>
```

To get specific elements in the output out of a set of sibling elements, an empty leaf node, called a *selection node*, is specified in the filter. For example, if you need the interface name and description for *all* three interfaces, the containment nodes <interface> and <GigabitEthernet> are used, and then the selection nodes <name> and <description> are used to construct the filter, as shown in Example 14-12.

Example 14-12 *Using the Selection Nodes <name> and <description> in a Subtree Filter*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <get-config>
        <source>
            <running />
        </source>
        <filter>
            <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
                <interface>
                    <GigabitEthernet>
                        <name></name>
                        <description />
                    </GigabitEthernet>
                </interface>
            </native>
        </filter>
    </get-config>
</rpc>]]>]]>
```

Note that an empty selection node can use one of two XML notations for empty elements. The first notation uses opening and closing tags with nothing in between, as the <name> element in the example does. The second notation uses just a single tag, as the <description> element does.

The result shown in Example 14-13 is a list of interfaces, with only names and descriptions included in the output.

Example 14-13 *Output Including Names and Descriptions Only for All Interfaces*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <GigabitEthernet>
                    <name>1</name>
                    <description>MANAGEMENT INTERFACE - DON'T TOUCH ME</description>
                </GigabitEthernet>
                <GigabitEthernet>
                    <name>2</name>
```

```
                <description>Network Interface</description>
            </GigabitEthernet>
            <GigabitEthernet>
                <name>3</name>
                <description>Network Interface</description>
            </GigabitEthernet>
        </interface>
    </native>
</data>
</rpc-reply>]]>]]>
```

What if you need to get the configuration for interface GigabitEthernet1 only but need the *full* interface configuration ? In this case, you need to use a *content match node*, which is a leaf node, but, unlike a selection node, it contains content used as the matching criteria.

Say that the leaf node <name> is to be used along with 1 as content. In this case, the filter retrieves all sibling elements having the hierarchy <native> → <interface> → <GigabitEthernet> that have a leaf node <name> containing the value 1. Example 14-14 shows the filter for this case.

Example 14-14 *Using a Content Match Node to Retrieve the Full Configuration of Interface GigabitEthernet1*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <get-config>
        <source>
            <running />
        </source>
        <filter>
            <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
                <interface>
                    <GigabitEthernet>
                        <name>1</name>
                    </GigabitEthernet>
                </interface>
            </native>
        </filter>
    </get-config>
</rpc>]]>]]>
```

The resulting rpc-reply message contains the full configuration of interface GigabitEthernet1, as shown in Example 14-15.

Example 14-15 *The Result of Using the Content Matching Node <name> to Display the Full Configuration of Interface GigabitEthernet1*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <GigabitEthernet>
                    <name>1</name>
                    <description>MANAGEMENT INTERFACE - DON'T TOUCH ME</description>
                    <ip>
                        <address>
                            <primary>
                                <address>10.10.20.48</address>
                                <mask>255.255.255.0</mask>
                            </primary>
                        </address>
                    </ip>
                    <mop>
                        <enabled>false</enabled>
                        <sysid>false</sysid>
                    </mop>
```

```
                <negotiation xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ethernet">
                    <auto>true</auto>
                </negotiation>
            </GigabitEthernet>
        </interface>
    </native>
</data>
</rpc-reply>]]>]]>
```

If the filter matches on a specific description instead of on the name, and if it happens that more than one interface have that same description, the full configuration of both interfaces appears in the output, since the <GigabitEthernet> elements of both interfaces are sibling elements.

Finally, if one of the elements in a data model has any attributes, the values of one or more of these attributes can be used as matching criteria for a subtree filter. This works just as a content matching node works: All sibling nodes that satisfy that specific attribute value show up in the filter results, along with all child elements of these siblings.

XPath Filters

An XPath filter uses XPath expressions to filter the data retrieved from a device. XPath is covered in detail in Chapter 10, and it is revisited here in the context of NETCONF.

To use an XPath filter, you add a <filter> element under the element representing the operation, with three distinct attributes:

• **type**: This attribute has the value **xpath**.

• **xmlns** : This attribute has a value equal to the namespace pointing to the data model referenced by the filter.

• **select** : This attribute has a value equal to the XPath expression that is used to filter the XML tree to retrieve only the data needed.

Example 14-16 shows an rpc message that uses an XPath filter to retrieve the <name> element of *all* interfaces on the router.

Example 14-16 *An rpc Message Using an XPath Filter to Retrieve the Names of All Interfaces on the Router*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <get-config>
        <source>
            <running />
        </source>
        <filter xmlns:xyz="http://cisco.com/ns/yang/Cisco-IOS-XE-native"
                type="xpath"
                select="/xyz:native/interface/GigabitEthernet/name">
        </filter>
    </get-config>
</rpc>]]>]]>
```

Example 14-17 shows the RPC reply message that contains the interface names.

Example 14-17 *The rpc-reply Message, Including the Names of All Interfaces on the Router*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <GigabitEthernet>
                    <name>1</name>
                </GigabitEthernet>
                <GigabitEthernet>
                    <name>2</name>
                </GigabitEthernet>
                <GigabitEthernet>
                    <name>3</name>
                </GigabitEthernet>
            </interface>
        </native>
    </data>
</rpc-reply>]]>]]>
```

Example 14-18 shows an rpc message that uses an XPath filter to retrieve the full configuration of the interface whose <name> element has the value 1 (interface GigabitEthernet1).

Example 14-18 *An RPC Message to Retrieve the Full Configuration of Interface GigabitEthernet1*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <get-config>
        <source>
            <running />
        </source>
        <filter xmlns:xyz="http://cisco.com/ns/yang/Cisco-IOS-XE-native"
                type="xpath"
                select="/xyz:native/interface/GigabitEthernet[name='1']">
        </filter>
    </get-config>
</rpc>]]>]]>
```

Example 14-19 shows the rpc-reply message that contains the interface configuration of interface GigabitEthernet1.

Example 14-19 *The RPC Reply Message That Includes the Full Configuration of Interface GigabitEthernet1*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <GigabitEthernet>
                    <name>1</name>
                    <description>MANAGEMENT INTERFACE - DON'T TOUCH ME</description>
                    <ip>
                        <address>
                            <primary>
                                <address>10.10.20.48</address>
                                <mask>255.255.255.0</mask>
                            </primary>
                        </address>
                    </ip>
                    <mop>
                        <enabled>false</enabled>
                        <sysid>false</sysid>
                    </mop>
                    <negotiation xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ethernet">
                        <auto>true</auto>
                    </negotiation>
                </GigabitEthernet>
            </interface>
        </native>
    </data>
</rpc-reply>]]>]]>
```

Changing Configuration: <edit-config>, <copy-config>, and <delete-config>

To change the configuration in a datastore on a device running a NETCONF server, three operations come into play: <edit-config>, <copy-config>, and <delete-config>.

The <edit-config> operation introduces changes to the configuration in a target datastore, using new configuration in the rpc message body, in addition to a suboperation that specifies how to integrate this new configuration with the existing configuration in the datastore. The <copy-config> operation is used to create or replace an entire configuration datastore. The <delete-config> operation is used to delete an entire datastore.

One of the <edit-config> operation parameters is the *target datastore*. This is the configuration datastore that the client wishes to edit, and it goes under the <target> element in the rpc message. The new configuration that the client needs to incorporate into the target datastore, referred to as the *source configuration*, typically goes under the <config> element in the rpc message (and there are more options here if the device supports the :url capability discussed later in this chapter). Example 14-20 shows an <edit-config> operation that changes the description under interface GigabitEthernet3.

Example 14-20 *Using the <edit-config> Operation to Change the Description on Interface GigabitEthernet3*

```
<rpc
    xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="3">
    <edit-config>
        <target>
            <running/>
        </target>
        <config>
            <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
                <interface>
                    <GigabitEthernet>
                        <name>3</name>
                        <description>New Description via NETCONF</description>
                    </GigabitEthernet>
                </interface>
            </native>
        </config>
    </edit-config>
</rpc>]]>]]>
```

Example 14-21 shows an rpc-reply message to a <get-config> rpc message, which confirms that the description was changed on the interface.

Example 14-21 *Confirming That the Description Was Changed on the Interface*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <GigabitEthernet>
                    <name>3</name>
                    <description>New Description via NETCONF</description>
                    <negotiation xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-ethernet">
                        <auto>true</auto>
                    </negotiation>
                </GigabitEthernet>
            </interface>
        </native>
    </data>
</rpc-reply>
```

The <edit-config> operation is actually a little more sophisticated than what you have just seen in the previous two examples. The <edit-config> operation has an attribute named *operation* that defines how the source configuration integrates with the target datastore configuration (sometimes referred to as the suboperation, since <edit-config> itself is referred to as an operation).

Say that you want to configure interface Loopback100 on the router—but only if the interface does not already exist. If it does, you do not wish to overwrite it. Let's take a look at the default behavior of <edit-config> and then look at how the do-not-overwrite requirement can be satisfied.

The default mode of <edit-config> is to merge the source and target configurations—that is, to overwrite the parts of the interface configuration that you have in the rpc message and leave the rest of the configuration intact. So, if you specify a new IP address for the interface in your rpc message, the router updates the interface IP address and leaves all the other sections of the interface configuration, such as the description, intact. Example 14-22 shows an rpc-reply message to a <get-config> operation that shows the current configuration of interface Loopback100.

Example 14-22 *Current Configuration on Interface Loopback100*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <Loopback>
                    <name>100</name>
                    <description>Testing NETCONF edit-config</description>
```

```
                    <ip>
                        <address>
                            <primary>
                                <address>10.10.10.10</address>
                                <mask>255.255.255.255</mask>
                            </primary>
                        </address>
                    </ip>
                </Loopback>
            </interface>
        </native>
    </data>
</rpc-reply>]]>]]>
```

The current IP address on the interface is 10.10.10.10, and the description is Testing NETCONF edit-config. In Example 14-23, the new IP address 10.20.20.20/32 is applied to the interface, and the interface description is not changed.

Example 14-23 *Using the <edit-config> Operation to Only Change the Interface IP Address*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
  <edit-config>
    <target><running/></target>
    <config>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <Loopback>
                    <name>100</name>
                    <ip>
                        <address>
                            <primary>
                                <address>10.20.20.20</address>
                                <mask>255.255.255.255</mask>
                            </primary>
                        </address>
                    </ip>
                </Loopback>
            </interface>
        </native>
    </config>
  </edit-config>
</rpc>]]>]]>
```

Example 14-24 shows the new interface configuration, with the new IP address and the same description as the old configuration.

Example 14-24 *New Interface Configuration: New IP Address and Same Description*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <Loopback>
                    <name>100</name>
                    <description>Testing NETCONF edit-config</description>
                    <ip>
                        <address>
                            <primary>
                                <address>10.20.20.20</address>
                                <mask>255.255.255.255</mask>
                            </primary>
```

```
                    </address>
                </ip>
            </Loopback>
        </interface>
    </native>
  </data>
</rpc-reply>]]>]]>
```

As you can see, the IP address is changed, and interface description is left intact. The new configuration in the rpc message and the interface configuration in the datastore are *merged*.

Now let's look again at the original requirement: If Loopback100 was already configured, and you attempt to change its configuration, you want to receive an error message and leave the current configuration intact. In this case, the create suboperation is used, instead of the default merge. In Example 14-25, an rpc uses the create operation to attempt to create interface Loopback100 with interface IP address 10.30.30.30, but as expected, you receive an error message, and the interface configuration is left as is.

Example 14-25 *The Create Suboperation*

```
! rpc message with an <edit-config> operation and "create" sub-operation>


<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
  <edit-config>
    <target><running/></target>
    <config>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native"
                xmlns:nf="urn:ietf:params:xml:ns:netconf:base:1.0">
            <interface>
                <Loopback nf:operation="create">
                    <name>100</name>
                    <description>Testing NETCONF edit-config</description>
                    <ip>
                        <address>
                            <primary>
                                <address>10.30.30.30</address>
                                <mask>255.255.255.255</mask>
                            </primary>
                        </address>
                    </ip>
                </Loopback>
            </interface>
        </native>
    </config>
  </edit-config>]]>]]>


! Error message received


<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
    <rpc-error>
        <error-type>application</error-type>
        <error-tag>data-exists</error-tag>
        <error-severity>error</error-severity>
        <error-path xmlns:ios="http://cisco.com/ns/yang/Cisco-IOS-XE-native"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    /nc:rpc/nc:edit-
config/nc:config/ios:native/ios:interface/ios:Loopback[ios:name='100']
  </error-path>
        <error-info>
```

```
            <bad-element>Loopback</bad-element>
        </error-info>
    </rpc-error>
</rpc-reply>]]>]]>


! interface configuration unchanged


<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="10">
    <data>
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <Loopback>
                    <name>100</name>
                    <description>Testing NETCONF edit-config</description>
                    <ip>
                        <address>
                            <primary>
                                <address>10.20.20.20</address>
                                <mask>255.255.255.255</mask>
                            </primary>
                        </address>
                    </ip>
                </Loopback>
            </interface>
        </native>
    </data>
</rpc-reply>]]>]]>
```

Notice that the <error-tag> value is data-exists. You should notice two things about the <edit-config> operation parameter. First, it is inserted at the element level, where the operation is required to take effect. Under the same <config> element you can use the merge operation with one element in the hierarchy and then use the create operation with another element in the hierarchy. How the different suboperations interact together when used simultaneously under the same <config> element is implementation specific; you need to do some experimenting to fully understand it.

The other thing to note is that the suboperations are always defined in the mandatory namespace urn:ietf:params:xml:ns:netconf:base:1.0, and this is why, in the previous examples, the namespace nf is declared in the element <native>, and then the operation attribute is defined inside that namespace.

<edit-config> supports the following operation values:

• **merge**: The NETCONF server analyses the source configuration and merges it with the configuration datastore at the same level in the data model hierarchy. This is the default value if no operation attribute is defined for an element.

• **replace**: The source configuration completely replaces the configuration in the target datastore at the same hierarchy level. If the configuration does not exist, it is created.

• **create**: The source configuration is added to the target datastore configuration with the same hierarchy as presented in the rpc message—if this new configuration did *not* exist in the target datastore configuration. If it did exist, an rpc-error is returned with the <error-tag> value data-exists.

• **delete**: The target configuration matching the source configuration element that has an operation attribute equal to delete is deleted from the datastore. If a matching element in the target datastore does not exist (that is, if you try to delete configuration that does not exist) an rpc-reply message with an <rpc-error> element is returned with the <error-tag> value data-missing.

• **remove**: This operation functions exactly like the delete operation except that if a matching element in the target datastore does not exist (that is, if you are trying to delete configuration that does not exist), no rpc-error is returned.

In addition to the operation parameter, the <edit-config> operation accepts three more parameters:

• **default-operation**: This parameter sets the default operation value, which is the value used if no operation attribute is defined for an element. This can be merge, replace, or none. In the case of none, no action is taken on the target datastore unless an operation attribute is explicitly defined.

• **test-option**: This parameter requires that the device support the :validate:1.1 capability (which is covered later in this chapter). It allows the network device to validate the configuration before the <edit-config> operation is executed. The three values allowed are test-then-set (that is, validation first, followed by execution), set (direct execution without validation), and test-only (validation only without execution).

• **error-option**: This parameter tells the network device what to do if it encounters an error condition while executing an <edit-config> operation. The three allowed values are stop-on-error, continue-on-error, and rollback-on-error. If an error condition is encountered while executing an <edit-config> operation, stop-on-error stops the execution without rolling back any changes. continue-on-error causes the device to record any error conditions encountered but continues executing the operation. rollback-on-error has the device stop the execution and roll back any changes made by the current <edit-config> operation.

If the <edit-config> operation is successful, an rpc-reply message is sent back to the client with an <ok> element. Otherwise an rpc-reply message with an appropriate <rpc-error> element is returned.

Next is the <copy-config> operation. Unlike the <edit-config> operation, the <copy-config> operation replaces *all* the contents of a target datastore with the contents of another *complete* source datastore. Therefore, the operation has only two parameters: <source> and <target>. If the :url capability is supported, remote datastores identified by a URI may be used as <source> or <target>. Restrictions on which datastores are allowed as source and which are allowed as target are implementation specific.

If the <copy-config> operation is successful, an rpc-reply message is sent back to the client with an <ok> element. Otherwise, an rpc-reply message with an appropriate <rpc-error> element is returned.

Finally, the <delete-config> operation is used to delete an entire <target> datastore. The exact meaning of deleting a datastore is implementation specific, but it generally means deleting the contents of the datastore. Which datastores a client is allowed to delete is also implementation specific. For example, some implementations do not support <delete-config> with any <target> datastore on the device. Only a remote datastore accessible via a URI may be acted on using this operation. However, the NETCONF protocol forbids deleting the <running> datastore for any implementation. Example 14-26 shows a typical rpc message with a <delete-config> operation.

Example 14-26 *An rpc Message with the <delete-config> Operation Acting on the Candidate Configuration as <target>*

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

    <delete-config>

        <target>

            <candidate/>

        </target>

    </delete-config>

</rpc>]]>]]>
```

If the operation is successful, an rpc-reply message with an <ok> element is returned. Otherwise, an rpc-reply message with an appropriate <rpc-error> element is returned.

Datastore Operations: <lock> and <unlock>

The <lock> operation is used to lock a specific datastore. Locking a datastore prevents any other entity from changing the contents of that datastore, whether it be another NETCONF session or a non-NETCONF entity such as a CLI session or SNMP. Locking the datastore also means that no other session can acquire a lock on that datastore until the current lock is released. A lock is released if the server receives an rpc message containing an <unlock> operation on the same session that issued the <lock> operation or if the session that owns the lock is terminated.

The <lock> and <unlock> operations have one mandatory parameter, which is the <target> datastore to be locked or unlocked. Example 14-27 shows two rpc messages, one to lock the candidate configuration datastore, and another to unlock it.

Example 14-27 *Two rpc Messages: One Calling the <lock> Operation, and the Other Calling the <unlock> Operation with the Candidate Configuration as the Target*

```
! lock Operation

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">

    <lock>

        <target>

            <candidate/>

        </target>

    </lock>

</rpc>]]>]]>


! unlock Operation

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="2">

    <unlock>

        <target>

            <candidate/>

        </target>

    <unlock>

</rpc>]]>]]>
```

If the <lock> or <unlock> operation is successful, an rpc-reply message containing an <ok> element is returned. Otherwise, the rpc-reply message contains an <rpc-error> element. A lock may fail for a number of reasons. For example, the datastore may be locked by another session. Or the target datastore may be the candidate, and while the datastore is not locked by another session, the candidate configuration may have been modified, and the modifications were neither committed nor discarded. You might want to experiment with the different error conditions and familiarize yourself with the different <rpc-error> elements received.

A session cannot unlock a configuration datastore that has been locked by another session. If a session attempts to unlock a datastore that it did not lock, it receives an error message that contains an <error-tag> element with the value lock-denied and an <error-info> element with a child <session-id> element containing the value of the session that owns the lock. The <session-id> value is an important value as it may be used in a <kill-session> operation to end the session holding the lock, as described in the next section.

Session Operations: <close-session> and <kill-session>

A NETCONF session is a persistent connection; that is, it should stay up as long as the client and server are exchanging rpc and rpc-reply

messages, and after that it may be terminated.

A NETCONF session can be explicitly terminated by a client via an rpc message with a <close-session> or <kill-session> operation, or it may be implicitly terminated by the server if certain conditions are triggered, such as the failure of the underlying transport or a session timeout due to inactivity.

The <close-session> operation gracefully terminates a session. That is, it completes any operation requested *before* the <close-session> operation, releases any locks held by the session, and tears down the transport connection with the client. The <close-session> operation has no parameters.

The <kill-session> operation, on the other hand, is used by a client to immediately terminate another session. When a <kill-session> is requested, the server aborts any operation in progress for that session, stops and rolls back any confirmed commit operations, and tears down the transport connection. However, it does not roll back any other changes made over the course of the session. The <kill-session> operation has one mandatory parameter, the <session-id> element, which has a value equal to the session ID of the session that is to be terminated. A session cannot request a <kill-session> operation on itself. If it does, an error is generated.

Both <close-session> and <kill-session> operations result in an rpc-reply message with an <ok> element if successful and an <rpc-error> element if not. Example 14-28 shows an example for each operation.

Example 14-28 *Two rpc Messages: One Calling the <close-session> Operation, and the Other Calling the <kill-session> Operation for <session-id> 12345*

```
! <close-session> operation

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">

    <close-session/>

</rpc>]]>]]>



! <kill-session> operation

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">

    <kill-session>

        <session-id>12345</session-d>

    </kill-session>

</rpc>]]>]]>
```

Candidate Configuration Operations: <commit>, <discard-changes>, and <cancel-commit>

When a device supports the :candidate capability, the client may call the two operations <commit> and <discard-changes>. When the :confirmed-commit capability is also supported, two more operations are added to the list: the <confirmed-commit> and <cancel-commit> operations. These capabilities are discussed in detail later in this chapter, in the section "NETCONF Capabilities."

Some devices support a candidate configuration datastore. The candidate configuration starts off as an exact copy of the running configuration. A user or NETCONF client applies configuration changes to the candidate configuration. In NETCONF, this is accomplished by setting the <target> element representing the target datastore to <candidate> under the <edit-config> or <copy-config> operations in the rpc message. The client then has the option to call the <discard-changes> operation to discard the changes to the candidate configuration such that it reverts to its initial state before the changes or to call a <commit> operation to copy the candidate configuration to the running configuration, effectively applying the changes to the running configuration.

Example 14-29 shows two rpc messages: one with a <commit> operation, and the other with a <discard-changes> operation.

Example 14-29 *Two rpc Messages: One Calling the <commit> Operation and the Other Calling the <discard-changes> Operation*

```
! commit operation

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">

    <commit/>

</rpc>]]>]]>



! discard-changes operation

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">

    <discard-changes/>

</rpc>]]>]]>
```

If the <commit> or <discard-changes> operations are successful, the server sends an rpc-reply message with an <ok> element. If they are not successful, the server sends back an rpc-reply message with the appropriate <rpc-error> element.

When a device also supports the :confirmed-commit capability (for example, Cisco devices running IOS XE Version 17.x and NX-OS Version 9.x), the client is allowed to call an extended <commit> operation in an rpc message called a *confirmed commit*, which is basically a conditional commit. To elaborate, the workflow for a confirmed commit is as follows:

1. The client implements changes to the candidate configuration and calls a <commit> operation that includes the <confirmed/> element

2. A timeout timer is started. The timer is configurable and is identified in NETCONF by the element <confirm-timeout>. The default value of this timer is 10 minutes on the latest versions of IOS XE and NX-OS.

3. If the timeout timer expires without any further intervention from the client, any changes made to the candidate configuration are rolled back

4. The client may issue three different operations before the timeout timer expires:

• A <commit> operation *without* the <confirmed/> option (called a *confirming* commit) that reflects the changes to the running configuration as a regular commit would

• Another confirmed commit operation using a <commit> with the <confirmed/> element that resets the timer to 10 minutes so that the countdown starts again

• A <cancel-commit> operation that discards the changes to the candidate configuration and cancels the confirmed commit operation altogether, without waiting for the timeout timer to expire

5. If the <persist> option is included under the confirmed commit operation, the candidate configuration changes may be committed (the confirming commit) by any session, not necessarily by the session that called the confirmed commit. The <persist> element is given a value in the confirmed commit operation. The other session that wishes to commit the changes is required to provide that value under the <persist-id> element of its <commit> operation.

Example 14-30 shows an rpc message calling the <commit> operation with the <confirmed/> element and the <persist> value set to 1234,XYZ. The <confirm-timeout> value is also set to 5 minutes.

Example 14-30 *An rpc Message Calling a Confirmed Commit Operation*

```
! confirmed-commit operation

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">

    <commit>

        <confirmed/>

        <confirm-timeout>300</confirm-timeout>

        <persist>1234,XYZ</persist>

    </commit>

</rpc>]]>]]>
```

Example 14-31 shows another session calling a confirming <commit> operation to commit the changes by the session in Example 14-30. Because the sessions are different (probably indicating two different clients), this client is required to use the value 1234,XYZ as the <persist-id> in order to be able to commit the changes.

Example 14-31 *An rpc Message Calling the <commit> Operation with a <persist-id> Value*

```
! confirming commit operation

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">

    <commit>

        <persist-id>1234,XYZ</persist-id>

    </commit>

</rpc>]]>]]>
```

Configuration Validation: <validate>

The <validate> operation checks the configuration in a datastore for syntax and semantic errors—that is, it *validates* that configuration. To be able to execute this operation, the device has to support the :validate capability, as explained in the section "NETCONF Capabilities," later in this chapter. Example 14-32 shows an rpc message using the <validate> operation to validate the running configuration.

Example 14-32 *An rpc Message Using the <validate> Operation to Validate the Running Configuration*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">

    <validate>

        <source>

            <running/>

        </source>

    </validate>

</rpc>]]>]]>
```

If the configuration is valid, an rpc-reply message is returned to the client with an <ok> element. If not, an rpc-reply message is returned with an appropriate <rpc-error> element.

The NETCONF Content Layer

The previous sections in this chapter use NETCONF messages starting with the root element. Which root element a message uses depends on the message type. The root element is followed by the operation element and then a number of child XML elements that provide the necessary element values for these operations to be executed in the manner intended by the client. What XML elements follow the operation element depend on the following factors:

• **Which operation is being used in the RPC**: Some elements are mandatory for some operations, such as the <source> element for the <get-config> operation.

• **Which YANG model is being referenced**: This defines the hierarchy of the elements that constitute the content in the message body, whether the message is an rpc or an rpc-reply message.

Chapter 13, "YANG," discusses how YANG is used to model device configuration and provide hierarchical templates to express configuration in a structured manner. This is exactly the function provided by the content layer.

To understand the correlation between the data model used and the XML hierarchy in the NETCONF message body, let's examine how the configuration of interface GigabitEthernet3 is expressed using two different models: the Cisco-IOS-XE-native and the ietf-interfaces YANG data models.

The node hierarchy from the ietf-interfaces model is shown on the left in Figure 14-3, and the actual running configuration from the router is shown on the right in the figure. Notice how the node hierarchy in the model maps to the running configuration hierarchy.

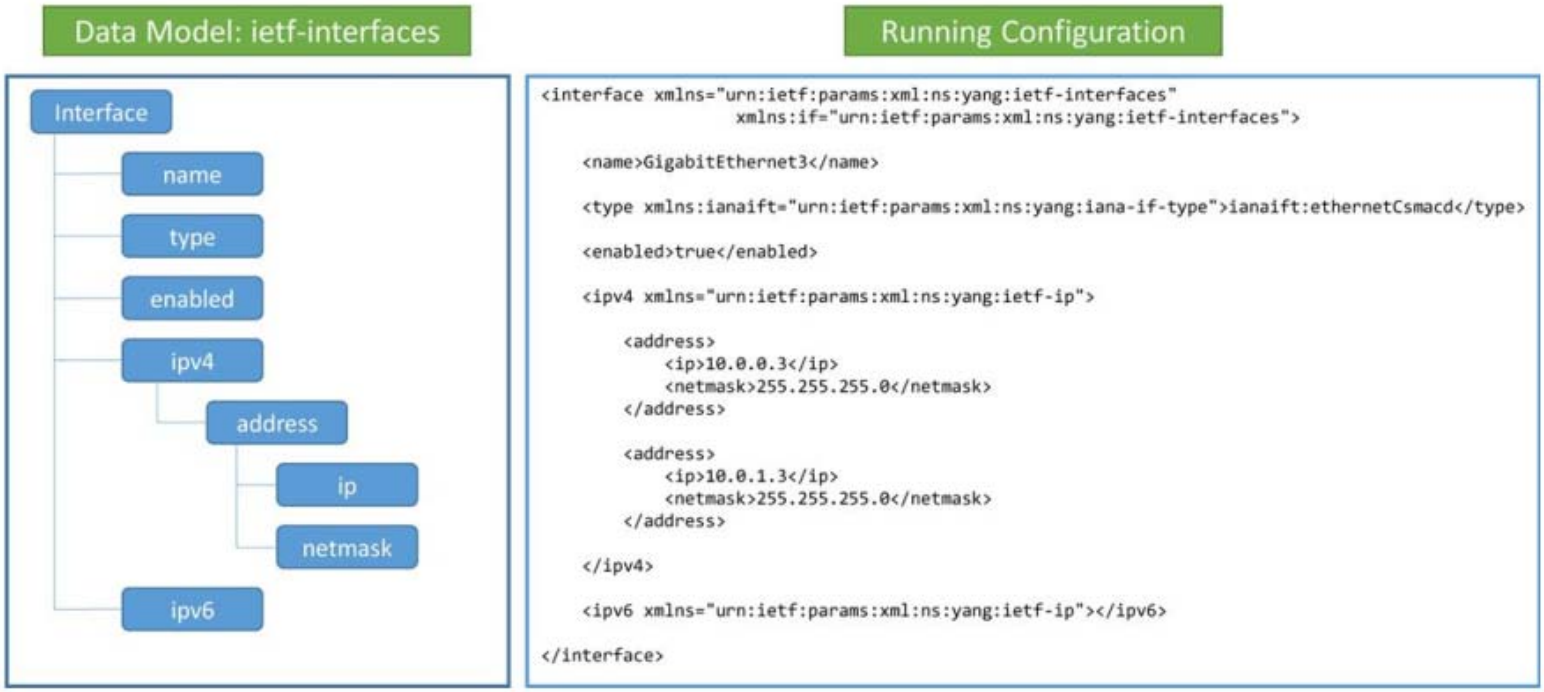

Figure 14-3 *The Configuration Hierarchy in the ietf-interfaces YANG Model Mapped to the Configuration on the Device*

The configuration shows up using this template because it was retrieved using an rpc message with a <get-config> operation that references this data model under the <filter> element, as highlighted in Example 14-33.

Example 14-33 *An rpc Message with a <get-config> Operation Referencing the YANG Model ietf-interfaces*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <get-config>
    <source><running/></source>
    <filter>
      <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
        <interface>
          <name>GigabitEthernet3</name>
        </interface>
      </interfaces>
    </filter>
  </get-config>
</rpc>]]>]]>
```

Now the same interface on the same router shows up with a different configuration hierarchy when referencing the YANG model Cisco-IOS-XE-native, as shown in Figure 14-4. Notice that all values are equal. The configuration only shows up structured differently.

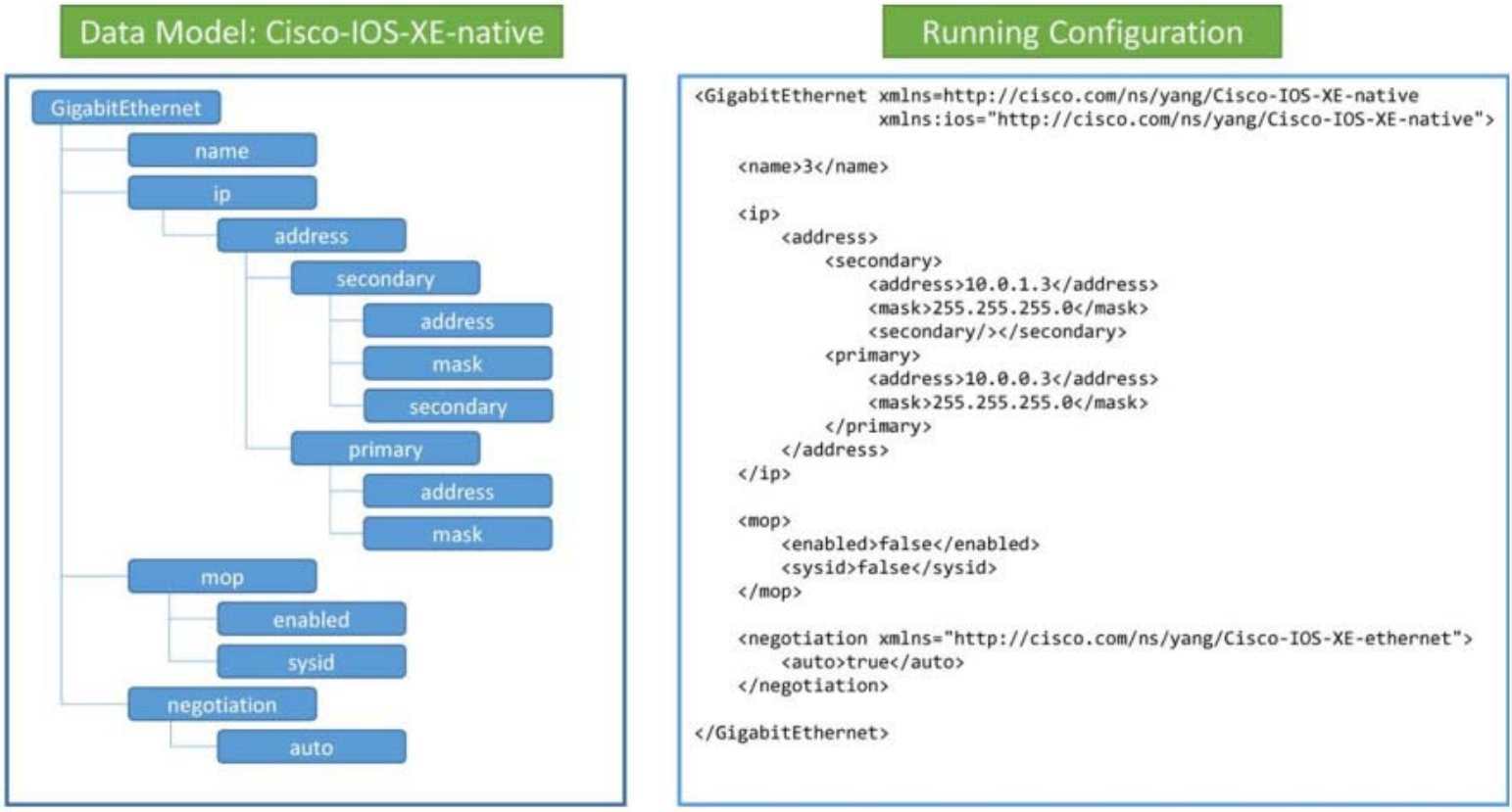

Figure 14-4 *The Configuration Hierarchy in the Cisco-IOS-XE-native YANG Model Mapped to the Configuration on the Device*

Again, the configuration shows up using this template because it was retrieved via an rpc message with a <get-config> operation that referenced the second data model under the <filter> element, as shown in Example 14-34.

Example 14-34 *An rpc Message with a <get-config> Operation Referencing the YANG Model Cisco-IOS-XE-native*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <get-config>
    <source><running/></source>
    <filter>
      <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
        <interface>
          <GigabitEthernet>
            <name>3</name>
          </GigabitEthernet>
        </interface>
      </native>
    </filter>
  </get-config>
</rpc>]]>]]>
```

In the previous examples, the data in the message body of the rpc-reply message that is returned by the server matches the YANG data model stated in the top-level element under the <filter> element in the rpc message. The same applies to rpc messages that use operations intended to edit the configuration, such as <edit-config>: The message body constituting the new configuration also matches a specific data model.

Now that the relationship between a data model and the message body has been established, how exactly is the data model referenced in the XML-encoded body of the message? As you have seen, the YANG data model is specified as the value of the attribute xmlns, which stands for *XML namespace*. Conceptually, each YANG module defines or creates a new namespace. The exact URI for that namespace is defined inside the YANG module. This URI is the value of the attribute xmlns in the XML message body and is an attribute for the top-level element in the module, which is <interfaces> in the case of the ietf-interfaces model and <native> in the case of the Cisco-IOS-XE-native model.

RFC 6022 specifies how to discover the YANG modules supported by a device: Send a <get> rpc message to the device with a child element <netconf-state> under <filter>, as in Example 14-35.

Example 14-35 *Discovering the YANG Modules Supported by a Device by Using the <get> Operation and the <netconf-state> Element*

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

```
    </get>
</rpc>
```

The resulting rpc-reply message lists all the YANG models (schemas) supported by the device, as shown in Example 14-36.

Example 14-36 *The List of YANG Models/Schemas Supported by a Device*

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <data>
        <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
            <schemas>
                <schema>
                    <identifier>ATM-FORUM-TC-MIB</identifier>
                    <version></version>
                    <format>yang</format>
                    <namespace>urn:ietf:params:xml:ns:yang:smiv2:ATM-FORUM-TC-
MIB</namespace>
                    <location>NETCONF</location>
                </schema>

--------- output omitted for brevity ---------

                <schema>
                    <identifier>Cisco-IOS-XE-interfaces</identifier>
                    <version>2019-03-11</version>
                    <format>yang</format>
                    <namespace>http://cisco.com/ns/yang/Cisco-IOS-XE-native</namespace>
                    <location>NETCONF</location>
                </schema>
                <schema>
                    <identifier>Cisco-IOS-XE-interfaces-oper</identifier>
                    <version>2018-10-29</version>
                    <format>yang</format>
                    <namespace>http://cisco.com/ns/yang/Cisco-IOS-XE-interfaces-
oper</namespace>
                    <location>NETCONF</location>
                </schema>

--------- output omitted for brevity ---------

                <schema>
                    <identifier>tailf-netconf-query</identifier>
                    <version>2017-01-06</version>
                    <format>yang</format>
                    <namespace>http://tail-f.com/ns/netconf/query</namespace>
                    <location>NETCONF</location>
                </schema>
            </schemas>
        </netconf-state>
    </data>
</rpc-reply>
```

To retrieve the actual schema content for one of the modules in the previous list, you can use the <get-schema> operation and use the <identifier>, <version>, and <format> values for that particular schema. Example 14-37 shows how to retrieve the schema contents for the Cisco-IOS-XE-interface module on the IOS XE sandbox.

Example 14-37 *The rpc Message for Retrieving the Schema Contents for a the Cisco-IOS-XE-interfaces Module Using the <get-schema> Operation*

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

```
    <get-schema xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
        <identifier>Cisco-IOS-XE-interfaces</identifier>
        <version>2019-03-11</version>
        <format>yang</format>
    </get-schema>
</rpc>
```

Example 14-38 shows the rpc-reply message that is received as a result.

Example 14-38 *The Contents of the Cisco-IOS-XE-interfaces Schema in the rpc-reply Message*

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101"><data
xmlns='urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring'><![CDATA[submodule Cisco-
IOS-XE-interfaces {
  belongs-to Cisco-IOS-XE-native {
    prefix ios;
  }
  import ietf-inet-types {
    prefix inet;
  }
  import Cisco-IOS-XE-types {
    prefix ios-types;
  }
  import Cisco-IOS-XE-features {
    prefix ios-features;
  }
  import Cisco-IOS-XE-interface-common {
    prefix ios-ifc;
  }
  organization
    "Cisco Systems, Inc.";
  contact
    "Cisco Systems, Inc.
     Customer Service
     Postal: 170 W Tasman Drive
     San Jose, CA 95134
     Tel: +1 1800 553-NETS
     E-mail: cs-yang@cisco.com";
  description
    "Cisco XE Native Interfaces Yang model.
     Copyright (c) 2016-2019 by Cisco Systems, Inc.
     All rights reserved.";
  // ================================================================
  // REVISION
  // ================================================================
  revision 2019-03-11 {
      description
      "Modified the leafref to leaf because of pyang issue";
  }
--------- output omitted for brevity ---------


      // interface Wlan-GigabitEthernet
      list Wlan-GigabitEthernet {
        description
          "WLAN GigabitEthernet";
        key "name";
```

```
        leaf name {
          type string;
        }
        uses interface-switchport-grouping;
        uses interface-common-grouping;
      }
    }
  }
}
]]></data>
</rpc-reply>]]>]]>
```

NETCONF Capabilities

As explained earlier, the first thing that happens when a transport session is complete is the exchange of hello messages between the server and client (in no particular order). A NETCONF hello message contains a list of sibling <capability> elements, each providing a URI that represents one capability supported by the device sending out the hello message.

Some capabilities are identified as base NETCONF capabilities and are defined using URNs. In the hello message in Example 14-1 earlier in this chapter, you can see some of these base capabilities at the very beginning of the hello message. Capabilities are an extensible feature of NETCONF, and different vendors or NETCONF implementations may define their own capabilities. In any case, a capability is always defined as a URI. The next few sections discuss the base NETCONF capabilities as defined in RFC 6241. These base capabilities always have this format:

urn:ietf:params:netconf:capability:{*name*}:{*version*}

where the *version* at the end of the URN identifies the version of the capability (*not* the version of NETCONF). The shorthand notation for the capability is **:**{*name*}**:**{*version*} or just **:**{*name*}.

Capabilities are used to state three classes of information by the peer sending the hello message:

• The NETCONF version supported by the peer. After the hello message exchange, the peers use the lowest version number agreed on by the two peers. The current version of NETCONF is 1.1, and the previous version is 1.0. As mentioned earlier in the chapter, the NETCONF version is indicated by the **urn:ietf:params:netconf:base:1.**{*x*} capability.

• The base and extended capabilities supported by the peer.

• The data models supported by the peer.

The Writable Running Capability

A device that supports the :writable-running capability allows the client to write directly to its running configuration. In NETCONF jargon, this means that the <target> element in an <edit-config> or <copy-config> operation can be <running>. The :writable-running capability is identified by the following URN:

urn:ietf:params:netconf:capability:writable-running:1.0

The Candidate Configuration Capability

A device that supports the :candidate capability has a candidate configuration datastore. When a device has a candidate datastore, the typical workflow involves the client implementing the configuration changes on the candidate configuration first and then issuing either a <commit> operation to copy the candidate configuration to the running configuration or a <discard-changes> operation to discard the changes made to the candidate configuration.

Support for this capability means that the device can accept the element <candidate> as the <source> or <target> datastore when the operation is <get-config>, <edit-config>, <copy-config>, or <validate>.

The candidate configuration may be shared between multiple sessions. Therefore, to avoid conflicting configuration changes from different sessions, the client should use the <lock> and <unlock> operations to lock the candidate configuration during configuration changes and then to unlock it when the lock is not needed anymore.

The :candidate capability is identified by the following URN:

urn:ietf:params:netconf:capability:candidate:1.0

The Confirmed Commit Capability

A device that supports the :candidate capability may also support the :confirmed-commit capability. A device that supports the :confirmed-commit capability allows the client to call a <confirmed-commit> operation in an rpc message. (The preceding section describes this operation.)

The :confirmed-commit capability is identified by the following URN:

urn:ietf:params:netconf:capability:confirmed-commit:1.1

The Rollback-on-Error Capability

A device that supports the :rollback-on-error capability supports the option to roll back any changes if an error occurs during an <edit-config> operation. To activate this option, you place the string rollback-on-error under the <error-option> element in the rpc message. The different error options for the <edit-config> operation are covered in the section "Changing Configuration: <edit-config>, <copy-config>, and <delete-config>," earlier in this chapter.

The :rollback-on-error capability is identified by the following URN:

urn:ietf:params:netconf:capability:rollback-on-error:1.0

The Validate Capability

The :validate capability means that a device is capable of executing the <validate> operation. This operation is covered in the section "Configuration Validation: <validate>," earlier in this chapter.

In addition, this capability extends the <edit-config> operation to accept the <test-option> element, discussed earlier in this chapter, in the section "Changing Configuration: <edit-config>, <copy-config>, and <delete-config>."

The :validate capability is identified by the following URN:

```
urn:ietf:params:netconf:capability:validate:1.1
```

The Distinct Startup Capability

If a device has a separate datastore for the startup configuration, the :startup capability allows the NETCONF client to operate on this startup configuration by using the element <startup/> either as <target> or <source>, depending on the operation. As is customary through the CLI, editing the running configuration does not automatically update the startup configuration. The running configuration needs to be explicitly copied to the startup configuration if the device is to boot up from the updated configuration. The same goes for NETCONF. If the running-configuration is updated, the changes are only reflected to the startup configuration after an explicit <copy-config> operation, where the <source> is <running/> and the <target> is <startup/>.

The :startup capability is identified by the following URN:

```
urn:ietf:params:netconf:capability:startup:1.0
```

The URL Capability

The :url capability allows a device to read and write to a remote XML document located at a URI (that is, a remote datastore).

The usage of this capability is slightly different for each operation. When it is used with the <edit-config> operation, the <config> element in the rpc message containing the configuration is replaced with a <url> element containing the link where the configuration resides. If you applied the configuration in Example 14-20 to a device by using a URI instead of by embedding it in the rpc message by using the <config> element, the rpc message would be similar to that shown in Example 14-39.

Example 14-39 *Using a URI and the :url Capability to Edit the Running Configuration*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
    <edit-config>
        <target>
            <running/>
        <target>
        <url>http://example.com/config-repo/config.xml<url>
    </edit-config>
</rpc>]]>]]>
```

In Example 14-39, the URI in the <url> element points to an XML document in which a <config> element contains the configuration in the mandatory namespace urn:ietf:params:xml:ns:netconf:base:1.0. Note that the URI does not have to point to a remote location over the Internet or even over the network. It could be a path to a file on the local file system.

When you use a URI with the <copy-config> operation, the <url> element resides under either the <source> element or the <target> element. In Example 14-40, a copy of the running configuration is backed up to a file on the local file system.

Example 14-40 *Backing Up the Running Configuration to a File on the Local File System*

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="1">
    <edit-config>
        <source>
            <running/>
        <source>
        <target>
            <url>file:///home/NetProg/Config_Backup_Latest<url>
        <target>
    </edit-config>
</rpc>]]>]]>
```

When the :url capability is used with the <delete-config> operation, the <url> element is always under the <target> element; when it is used with the <validate> operation, it is always under the <source> element.

The :url capability is identified by the following URN:

```
urn:ietf:params:netconf:capability:url:1.0?scheme=name-1[,name-2[,...][,name-n]
```

The URI for this capability includes a scheme argument that lists all the schemes supported by the device. The following is an example of a valid :url capability URI:

```
urn:ietf:params:netconf:capability:url:1.0?scheme=http,ftp,file
```

The XPath Capability

In the section "Retrieving Data: <get> and <get-config>," earlier in this chapter, you saw that a filter can be applied to the <get> and <get-config> operations in order to select specific nodes, as per a specific data model, to be returned in the rpc-reply message. If a filter is not

applied, all configuration or configuration and state data is returned. A filter may be one of two types: subtree or XPath. The capability :xpath allows a device to apply filters of type XPath to the <get> and <get-config> operations, as described earlier in this chapter.

The :xpath capability is identified by the following URN:

```
urn:ietf:params:netconf:capability:xpath:1.0
```

NETCONF Using Python: ncclient

So far in this chapter, you have seen how to construct NETCONF messages manually and send them to a network device (also manually) over an SSH connection. It is important to transition through this manual stage in order to fully understand the protocol. However, this manual method of using NETCONF is not scalable and is actually counterintuitive, considering that this book is about automation. This section shows how to use Python to communicate with a device via NETCONF by using the ncclient Python library.

The ncclient library provides the facility of emulating a NETCONF client using Python while masking some of the finer details of the protocol so that the programmer does not have to deal directly with most of the protocol specifics. This level of abstraction, which is expected from a high-level programming language library, lets the programmer focus on the task at hand instead of having to worry about things like the syntax and semantics of NETCONF.

The developers of ncclient claim to support all the functions of NETCONF covered in the older RFC 4741. Moreover, ncclient is extensible; that is, new transport protocols and protocol operations can be added to the library when needed. ncclient also has focused support for particular vendors, with, of course, Cisco at the very top of the list.

To use ncclient, you need to install the ncclient library. First, you install the following list of dependencies (using **yum** or **dnf** if you are on a CentOS or RHEL box):

```
setuptools 0.6+

Paramiko 1.7+

lxml 3.3.0+

libxml2

libxslt

libxml2-dev

libxslt1-dev
```

Then you need to download the Python script setup.py from the GitHub repo https://github.com/ncclient/ncclient and run it:

```
[NetDev@localhost ~]$ sudo python setup.py install
```

Or you can use pip to install ncclient:

```
[NetDev@localhost ~]$ sudo pip install ncclient
```

The ncclient library operates by defining a handler object called *manager* that represents the NETCONF server. The manager object has different methods defined to it, each performing a different protocol operation. To better understand ncclient, read through Example 14-41, and most of it will make sense right away.

Example 14-41 *Sending an rpc Message with a <get-config> Operation Using ncclient*

```
from ncclient import manager

     filter_loopback='''
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <Loopback>
                    <name>100</name>
                </Loopback>
            </interface>
        </native>
       '''

with manager.connect(host='ios-xe-mgmt-latest.cisco.com',
                     port=10000,
                     username='developer',
                     password='C1sco12345',
                     hostkey_verify=False
                     ) as m:

     rpc_reply = m.get_config(source="running",filter=("subtree",filter_loopback))

     print(rpc_reply)
```

In the Python script in Example 14-41, the manager module is first imported from ncclient. A subtree filter is defined as a multiline string named filter_loopback to extract the configuration of interface Loopback100 from the router. The filter follows the same syntax rules as the subtree filters constructed earlier in this chapter.

A connection to the router is then initiated using the manager.connect method. The parameters passed to the method in this particular example use values specific to one of Cisco's IOS XE sandboxes. The parameters are the host address (which may also be an IP address), the port configured for NETCONF access, the username and password, and hostkey_verify.

When a client connects to an SSH server, the server key is stored on the client, typically in the ~/.ssh/known_hosts file. If hostkey_verify is set to True, one of the keys stored on the client has to match the key of the server that the client is trying to connect to via SSH. When set to False, the

SSH keys on the client are not verified. For convenience, the hostkey_verify value is set to False in this example.

Then the get_config method, using the defined subtree filter, and parameter source with the value running, retrieves the required configuration from the running configuration datastore. The get_config method is also capable of using xpath filters.

Finally, the rpc-reply message received from the router is assigned to string rpc_reply and printed out. Example 14-42 shows this message.

Example 14-42 *The rpc-reply Message Back from the Server Containing Interface Loopback100's Configuration*

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-
id="urn:uuid:ed4c62a8-10be-47ad-b8ef-8a468212a9b5"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
      <interface>
        <Loopback>
          <name xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">100</name>
          <description>Testing ncclient</description>
          <ip>
            <address>
              <primary>
                <address>10.10.10.10</address>
                <mask>255.255.255.255</mask>
              </primary>
            </address>
          </ip>
        </Loopback>
      </interface>
    </native>
  </data>
</rpc-reply>
```

The manager.connect and get_config methods have a few more parameters that may be used for more granular control of the functionality. Only the basic parameters are covered here.

Similarly, the edit_config method can be used to edit the configuration on the routers. In Example 14-43, the edit_config method is used to change the IP address on interface Loopback100 to 10.20.20.20/32.

Example 14-43 *Using the edit_config Method to Change the IP Address on Interface Loopback100*

```
from ncclient import manager
     config_data='''
       <config>
         <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
           <interface>
             <Loopback>
               <name>100</name>
               <ip>
                 <address>
                   <primary>
                     <address>10.20.20.20</address>
                     <mask>255.255.255.255</mask>
                   </primary>
                 </address>
               </ip>
             </Loopback>
           </interface>
         </native>
       </config>
       '''
with manager.connect(host='ios-xe-mgmt-latest.cisco.com',
```

```
                    port=10000,
                    username='developer',
                    password='C1sco12345',
                    hostkey_verify=False
                    ) as m:
    rpc_reply = m.edit_config(target="running",config=config_data)
    print(rpc_reply)
```

The difference between the get_config and edit_config methods is that the latter requires a config parameter instead of a filter parameter, represented by the config_data string, and requires a target datastore instead of a source datastore.

Example 14-44 shows the output after running the script in Example 14-43, which is basically an rpc-reply message with an <ok> element. The **show run interface Loopback100** command output from the router shows the new interface configuration.

Example 14-44 *Results of Sending the rpc Message from Example 14-43*

```
! Output from the NETCONF Session
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-
id="urn:uuid:7da14672-68c4-4d7e-9378-ad8c3957f6c1"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <ok />
</rpc-reply>


! Output from the router via the CLI showing the new interface configuration
csr1000v-1#show run interface lo100
Building configuration...


Current configuration : 99 bytes
!
interface Loopback100
 description Testing ncclient
 ip address 10.20.20.20 255.255.255.255
end
```

This section provides only a brief introduction to the ncclient library. For a more comprehensive coverage, check out the documentation at https://ncclient.readthedocs.io/en/latest/#.

RESTCONF

RESTCONF is often referred to as the RESTful version of NETCONF because RESTCONF is compatible with NETCONF, and because specific mappings exist between several of the NETCONF and RESTCONF protocol features. However, this is not entirely true due to the inherent differences between the RPC-based API architecture on which NETCONF is based, and the REST framework on which RESTCONF is based. Also, RESTCONF implements only a subset of the NETCONF functionality. This section discusses RESTCONF in detail, as well as how it relates to NETCONF.

Note

RESTCONF is a RESTful protocol, and it would be a good idea to review Chapters 7, "HTTP and REST," and 8, "Advanced HTTP," before you start with RESTCONF, to get a better understanding of the protocol.

Protocol Overview

REST is the dominant framework for software engineering, and it was inevitable that someone would come up with a RESTful protocol that is compatible with NETCONF. In January 2017, the NETCONF working group in the IETF released RFC 8040, "RESTCONF Protocol." The number of RFCs covering RESTCONF is quite limited compared to the plethora of RFCs covering NETCONF. As of this writing, the list of RFCs covering RESTCONF is short:

• RFC 8040, "RESTCONF Protocol"

• RFC 8071, "NETCONF Call Home and RESTCONF Call Home"

• RFC 8527, "RESTCONF Extensions to Support the Network Management Datastore Architecture"

RESTCONF is a client/server protocol based on HTTP, so unlike NETCONF, which is based on SSH, it is not session based. Recall from Chapter 7 that the second constraint for REST APIs is that an API should be *stateless*.

A RESTCONF transaction is a regular HTTP message exchange involving a client composing and sending an HTTP request message that contains all information pertaining to that particular transaction. The server then processes the request and replies with an HTTP response message. After the response message is sent out, the server maintains no state information for that client or that transaction.

As you saw in Chapter 7, an HTTP message is basically a start line, a number of headers, and the message body. The start line of a *request* message (from the client) includes the HTTP method, a URI segment pointing to the resource (or its representation) on which the method should act, and the HTTP version used; it should look similar to:

GET /restconf/data?content=config HTTP/1.1

The start line of a *response* message includes the HTTP version, a status code indicating the result of processing the request, and a status description or text that elaborates on the status code, such as:

HTTP/1.1 200 OK

As a RESTful protocol, RESTCONF is like all the other RESTful protocols in the following respects:

• It uses HTTP for transport.

• It uses HTTP methods to perform create, read, update, and delete (CRUD) operations. These HTTP methods map to NETCONF's RPC operations.

• It uses hierarchical URIs to identify resources that represent manageable components on the server on which it is running.

• It uses headers and a message body to communicate to the server the metadata and data, respectively, that pertain to a specific transaction. An example of metadata would be the value of the Accept header in which the client indicates to the server whether it can accept data in JSON or XML format. An example of data would be the configuration that the client wishes to add to a datastore in a POST request message.

In order to understand RESTCONF and effectively work with it, you need to understand what resources RESTCONF uses and how to use HTTP methods to retrieve and manipulate data on the RESTCONF server (the network device, in this case).

Whereas NETCONF only supports XML encoding, RESTCONF supports both XML and JSON. NETCONF supports models expressed in XSD and YANG, while RESTCONF supports only YANG models.

Note

RFC 7951 defines JSON encoding of data modeled with YANG.

RESTCONF does *not* have the capability to work with multiple datastores. It works with a single *conceptual* datastore. When RESTCONF is used to edit a configuration, this conceptual target datastore may be the candidate or the running configuration, depending on two factors:

• What datastores are present or supported by the device

• Whether the device is also running NETCONF in addition to RESTCONF

RESTCONF interacts with NETCONF based on a set of rules:

• If the device is not running NETCONF, RESTCONF applies edits directly to the running configuration of the device.

• If the device is running a NETCONF server besides the RESTCONF server, then only if the NETCONF server supports the :writable-running capability are the edits made by RESTCONF applied directly to the running configuration.

• If the NETCONF server supports the :candidate capability, then the edits performed by RESTCONF are applied to the candidate configuration instead. However, RESTCONF issues an automatic and immediate commit after the configuration edits are done in order to reflect the changes to the running configuration.

• If the NETCONF server supports the :startup capability, the RESTCONF edit operation updates the startup configuration, also automatically and immediately, after the configuration (running or candidate) is edited.

Note

These rules are defined in Section 1.4 of RFC 8040. However, whether an equipment vendor chooses to follow these rules (and implement a standards-based version of RESTCONF/NETCONF) or not is really up to the vendor. The devices running IOS XE that were used to generate the examples for this chapter do, in fact, behave according to these rules.

While RESTCONF understands the concept of configuration locking, it cannot manipulate locks. So, if a RESTCONF client attempts to manipulate the configuration on a network device and the datastore is locked by a NETCONF server on the same device, the RESTCONF server responds to the client with an appropriate error message, typically an HTTP response message with a “409 Conflict” status code. (Remember that RESTCONF is compatible with NETCONF.)

By no means is RESTCONF intended to replace NETCONF. RESTCONF is the go-to protocol when a client wishes to access a server (device) programmatically via a RESTful API, such as web applications used to push configuration to other components in the automation toolchain. However, the constraints placed on RESTful APIs limit the use cases and applicability of these APIs to one-to-one transactions (that is, a single client to a single network device, controller, or orchestrator).

NETCONF, on the other hand, with its session-based architecture, support for different datastores, and configuration validation capabilities (recall the :validate capability and <validate> operation), is more suitable for one-to-many transactions, where a single client is used to push configuration to several network devices. This is why products such as Cisco's NSO use NETCONF when configuring network devices in the southbound direction and use RESTful APIs in the northbound direction when communicating with web portals, event management systems, or ticketing systems.

Protocol Architecture

Figure 14-5 illustrates the four-layer model of the RESTCONF protocol, representing its functional components.

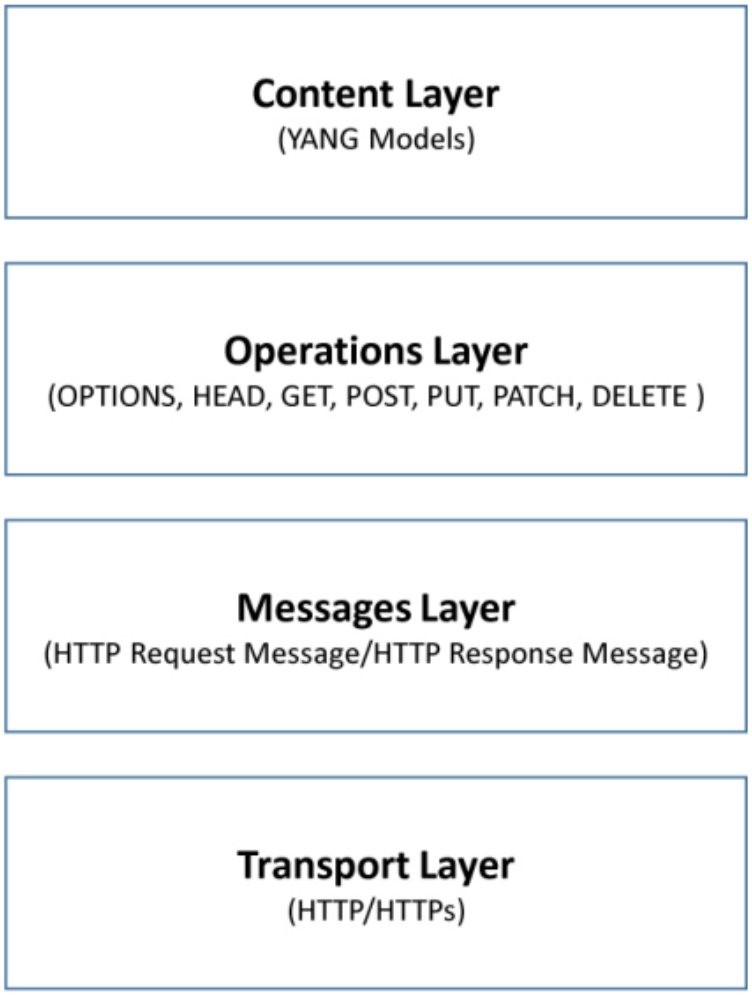

Figure 14-5 *The Four-Layer Model Encompassing All RESTCONF Functions*

The transport layer at the bottom represents the functions performed by the transport protocol used by RESTCONF—that is, HTTP. HTTP and RESTCONF are discussed in the section "The RESTCONF Transport Layer."

On top of the transport layer is the messages layer. The fact that RESTCONF uses HTTP implies that the messages used by RESTCONF are regular HTTP request and response messages. HTTP messages address resources, which represent manageable components on the device, by using URIs. HTTP messages and RESTCONF resources are discussed in the section "The RESTCONF Messages Layer."

The next layer is the operations layer. RESTful protocols like RESTCONF that are based on HTTP use HTTP methods exclusively to act on resources. The subset of HTTP methods used in RESTCONF and how these methods map to the NETCONF operations are discussed in the section "Methods and the RESTCONF Operations Layer."

The top layer of the model, the content layer, contains the same functional components of NETCONF, the YANG data models that represent the hierarchy, and characteristics of the data that are in the HTTP message body. Whatever applies to NETCONF applies to RESTCONF, except that RESTCONF does not support any schema description or modeling languages except YANG. As mentioned earlier, this YANG modeled data may be encoded in XML or JSON format. Refer to the earlier section "The NETCONF Content Layer" for a discussion of YANG models and their relationship with the content in the message body.

The RESTCONF Transport Layer

RESTCONF uses HTTP as the transport protocol. Strictly speaking, the protocol specification mandates the use of HTTP over TLS, which is more commonly known as HTTPS. TLS and HTTPS are covered extensively in Chapter 8.

While RESTCONF delegates data confidentiality and integrity to HTTPS, the authentication process performed by HTTPS results in a value that is passed on to RESTCONF, referred to as the *RESTCONF username*, which RESTCONF uses to authorize subsequent user request messages.

All other aspects of the RESTCONF transport layer are no different from the normal operations of HTTP and HTTPS, as covered in Chapters 7 and 8.

The RESTCONF Messages Layer

RESTCONF is a RESTful protocol based on HTTP. Therefore, a RESTCONF client sends standard HTTP request messages to a RESTCONF server, and the server normally responds with an HTTP response message. The format of RESTCONF messages are no different than the format of any other HTTP messages (covered in detail in Chapter 7). For your convenience, HTTP/RESTCONF message formats are reviewed briefly in the next two sections.

Request Messages

Figure 14-6 shows the format of a typical RESTCONF request message.

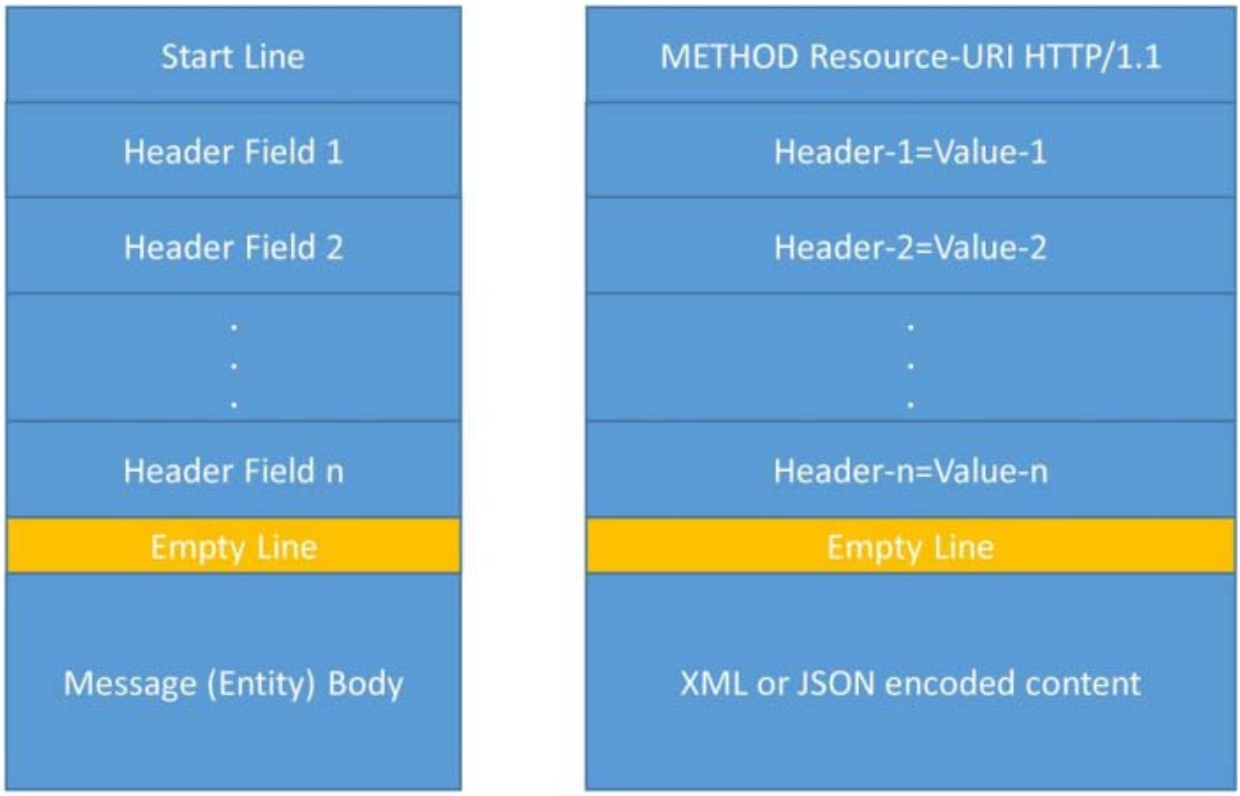

Figure 14-6 *Format of a Typical RESTCONF Request Message Matching a Typical HTTP Request Message*

An HTTP request message is composed of a start line, followed by a list of headers, an empty line marking the end of the headers section, and finally the message body.

The start line of an HTTP request message is composed of three values: the HTTP method, a URI segment representing the path to the resource, and the HTTP version, which is almost always HTTP/1.1.

Each header is each formatted as {*header-name*}={*header-value*}, and each of them is on a separate line. Headers appear in a message in no particular order.

The message body is composed of XML- or JSON-encoded content. The encoding used in the body is indicated using the Content-Type header field. Two Content-Type values that you will come across very often are application/yang-data+xml and application/yang-data+json. The encoding that the client would like to receive back in the server response message is indicated in the Accept header field of the request and uses the same type values as the Content-Type header. The message body is optional.

Response Messages

Figure 14-7 shows the format of a typical RESTCONF response message.

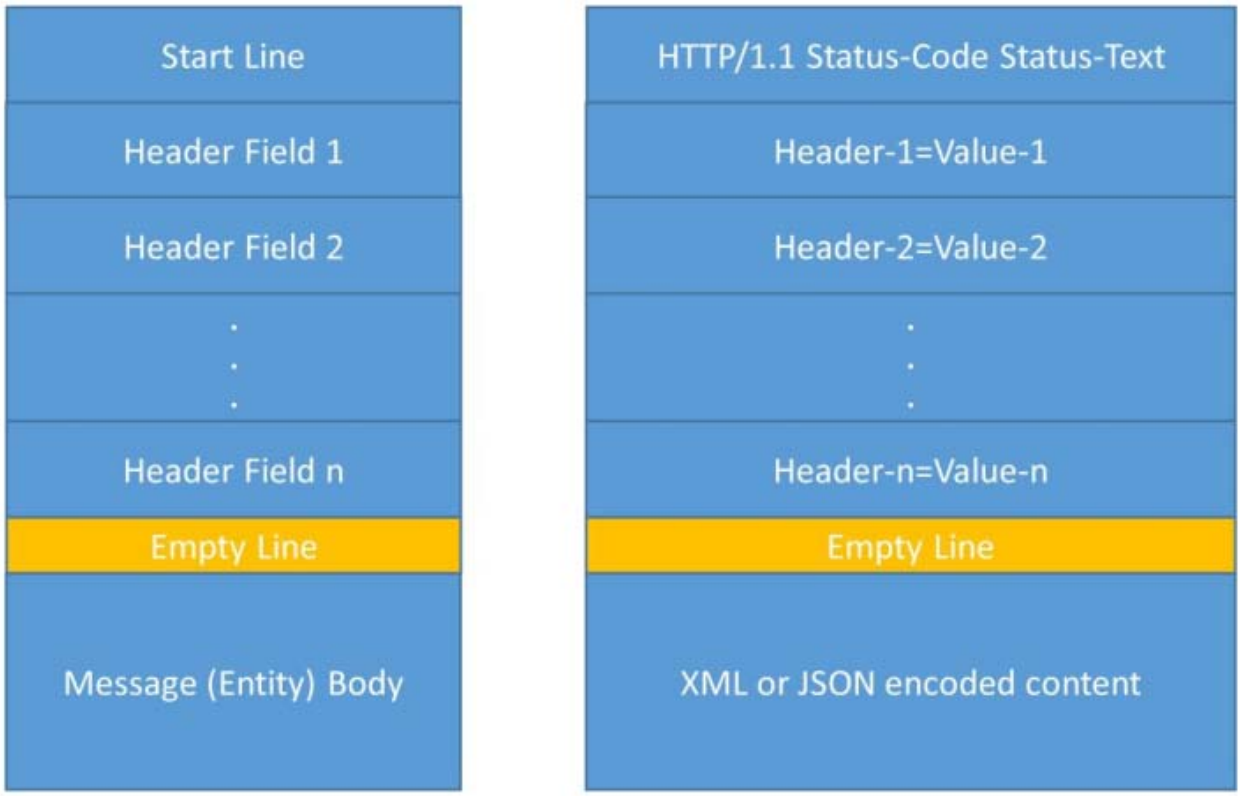

Figure 14-7 *Format of a Typical RESTCONF Response Message Matching a Typical HTTP Response Message*

The only difference between the format of a request message and a response message is in the start line. The start line of a response message starts with the HTTP version, followed by the status code and then status text, both of which give brief information on the status of

processing the client request. The headers section and message body are formatted exactly as in request messages.

Constructing RESTCONF Messages

To construct RESTCONF request messages successfully and understand response messages, you need to know the following:

• How to build the resource URI correctly in order to target the resource that you want to act on. This is the subject of the section "Resources."

• What method to use, coupled with the resource URI, to define what the RESTCONF operation will be. This is the subject of the section "Methods and the RESTCONF Operations Layer."

• What headers to include in the message and what their values should be. HTTP headers are comprehensively covered in Chapter 7. The use of HTTP headers in RESTCONF is covered in the next section, "RESTCONF HTTP Headers."

• What needs to go into the message body—that is, the JSON- or XML-encoded content, which depends on the YANG module referenced by the content. This is covered in the section "The NETCONF Content Layer."

RESTCONF HTTP Headers

HTTP has a huge number of defined headers that are used to communicate resource metadata back and forth between the client and the server. This section discusses the header fields that are of special relevance to RESTCONF.

A RESTCONF message body is encoded in either XML or JSON. The client and server indicate the encoding of the data in their messages by using the Content-Type header field. If the server receives an unsupported media type from the client, it responds with a "415 Unsupported Media Type" response message.

The client expresses the preferred encoding to be used in the response from the server by using the Accept header field. The server typically responds using the preferred encoding. If the preferred encoding is not supported by the server, the server responds with a "406 Not Acceptable" response message. If the Accept header is not used at all, the server either uses the encoding the client used in the request message or, if the request did not have content, the server uses an arbitrary default encoding that varies by implementation.

If a client request results in a new resource being created, the path to the new resource is identified by the Location header field in the response message.

A server responding to a retrieval request for a resource includes a Last-Modified header field, whose value is a timestamp indicating the last time the resource was modified. The client may use the If-Modified-Since or the If-Unmodified-Since header field in its request to edit a resource only if it was *changed* or *not changed*, respectively, since a specific time. This ensures that the client is not changing a resource that has been changed by another client between the time indicated in the header field and the time the client sent its request. This, of course, applies to configuration resources only, not to state/operation resources.

A resource representation also has a unique entity tag that changes each time the resource representation is updated. This tag is communicated to the client in the ETag header field. As with the last-modified timestamp, a client can make use of the If-Match or the If-None-Match header field in a request message to make sure that the entity tag has *not changed* or *changed*, respectively, before the server edits a resource. Again, this only applies to the configuration resources, not to state/operation resources.

The Last-Modified timestamp and ETag value are also maintained for the datastore and change when the contents of the datastore change. As you will read later in this chapter, the datastore itself is a resource; therefore, each time any configuration item is edited, the Last-Modified timestamp and ETag values for the datastore resource are changed, as are the values belonging to the resource that was edited. As a matter of fact, because resources are hierarchical, when a resource is edited, the two values belonging to all resources up the hierarchy, all the way to the datastore resource, are updated as well.

RESTCONF Error Reporting

Errors in RESTCONF are communicated back to the client by using HTTP status codes in response messages. When the status code returned by a server is a 4xx or 5xx code, this is considered an error condition, and an <error> element is included in the body of the message. The <error> elements contain the same information that would be included in the <rpc-error> element if that same error happened due to a NETCONF operation. The same error fields discussed earlier in this chapter, in section "RPC Reply Messages," are included as child elements to the <error> element in the RESTCONF response message.

Resources

A resource represents one manageable component in a device. In any HTTP-based protocol such as RESTCONF, a resource is identified by a URI. Therefore, the first step in constructing and understanding HTTP/RESTCONF messages is to be able to understand how URIs map to resources or, more accurately, resource representations. In the coming paragraphs, you will build fully functional URIs starting with the URI scheme.

As you have already read in this chapter, RESTCONF is based on HTTPS. Therefore, any RESTCONF URI starts with the scheme https://. Then comes the device IP or address and port, resulting in: **https://***{device_address}***:***{port}***/**. For example, the URI for reaching one of Cisco's IOS XE sandboxes is https://ios-xe-mgmt-latest.cisco.com:9443/, where 9443 is the (non-default) port for RESTCONF configured on the device. (See the note in the "NETCONF over SSH" section, earlier in the chapter, regarding the Cisco DevNet sandboxes.) The default port for RESTCONF is the HTTPS default port 443. Before proceeding with a detailed discussion of RESTCONF resources, take a look at the resources hierarchy in Figure 14-8 and how it relates to the resource URI.

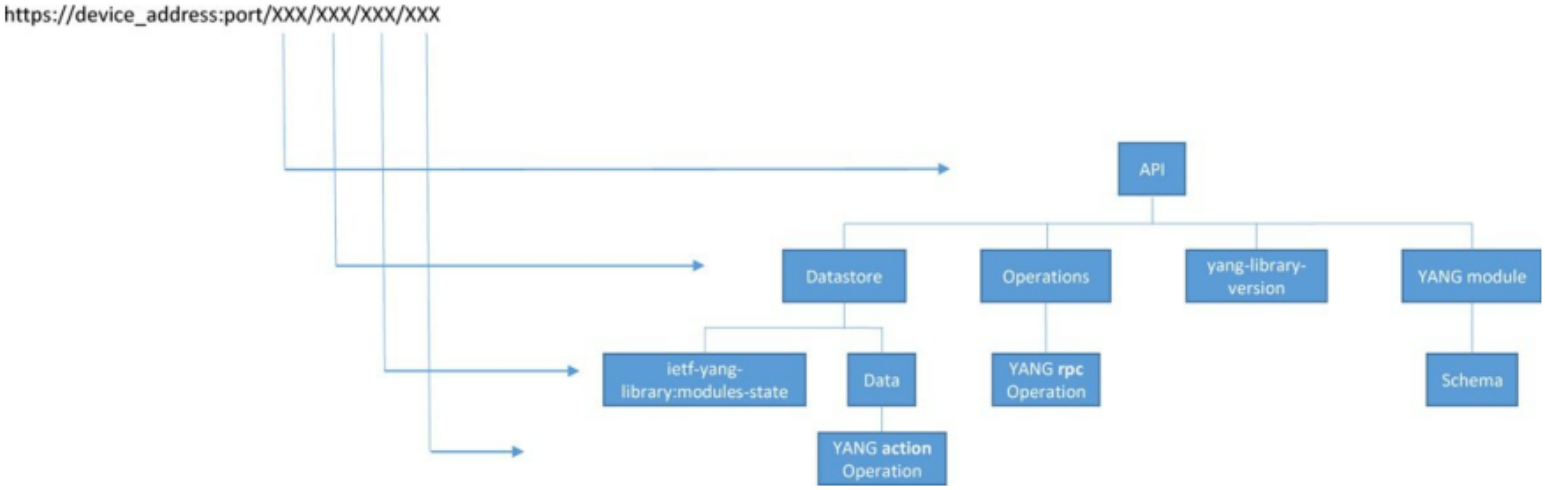

Figure 14-8 *RESTCONF Resources and Their Relationship with the Resource URI*

The API Resource

So far, the scheme, device, and RESTCONF port have been identified, but not any resources, yet. The first resource defined in RESTCONF is the protocol API itself, called the *root* or *API resource*, and it is the top-level resource in the resources hierarchy. As you can see in Figure 14-8, the API resource is at the top of the tree. When we add this to the resource URI, the URI becomes **https://**{*device_address*}**:**{*port*}**/**{*api_resource*}.

What is the value that needs to go into the URI to access the API resource? This question is answered in RFC 6415, "Web Host Metadata." The RFC states that performing a GET request to the ./well-known/host-meta URI retrieves the host-meta document formatted in XRD 1.0 (Extensible Resource Document) format. This host-meta document has the answer to this question. A GET request is sent to the Cisco IOS XE sandbox at https://ios-xe-mgmt-latest.cisco.com:9443/.well-known/host-meta using Postman, and the result is shown in Example 14-45.

Example 14-45 *GET Request to the /.well-known/host-meta URI to Retrieve the Value of the API Resource*

```
! GET request to /.well-known/host-meta

GET /.well-known/host-meta HTTP/1.1

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

Connection: keep-alive


! host-meta Document back from the server

HTTP/1.1 200 OK

Server: nginx

Date: Fri, 22 Nov 2019 12:06:11 GMT

Content-Type: application/xrd+xml

Content-Length: 107

Connection: keep-alive

Vary: Accept-Encoding


<XRD xmlns='http://docs.oasis-open.org/ns/xri/xrd-1.0'>

  <Link rel='restconf' href='/restconf'/>

</XRD>
```

In the response message body in Example 14-45, you can see the root <XRD> element and its child, the <Link> element, under it. Each <Link> element, according to RFC 6415, "conveys a link relation between the host described by the document and a common target URI." Simply put, the <Link> element in this example provides the URI /restconf/ (using the attribute href) as a link to a resource described as restconf (using the attribute rel) residing on the host on which HTTP is running. This is the path segment pointing to the API resource on that device. Applying this to the Cisco IOS XE sandbox, the API resource is identified by the URI https://ios-xe-mgmt-latest.cisco.com:9443/restconf.

Because the API resource is, in fact, a resource, sending a GET request to its URI should retrieve a representation of this resource. This is exactly what Example 14-46 shows—again using Postman.

Example 14-46 *GET Request to the API Resource*

```
! GET request to the API Resource

GET /restconf/ HTTP/1.1

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

Connection: keep-alive


! A representation of the API Resource back from the server

HTTP/1.1 200 OK

Server: nginx

Date: Fri, 22 Nov 2019 12:26:41 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


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

  <data/>

  <operations/>

  <yang-library-version>2016-06-21</yang-library-version>

</restconf>
```

As you can see from the response message in Example 14-46, the API resource has three child (second-level) resources:

- The datastore (data) resource
- The operations (operations) resource
- The YANG library version (yang-library-version) resource

The Datastore Resource

The datastore resource represents the datastore on the device. Remember that RESTCONF defines and works with only one datastore. The datastore resource is the parent to all configuration and state data resources on the device. The datastore resource is always identified by the path segment /data. This means that in order to access the datastore resource, the resource URI becomes **https://**{*device_address*}**:**{*port*}/{*api_resource*}/**data**. In the case of the IOS XE sandbox, the URI for the datastore resource is https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data.

A GET request sent to the datastore resource retrieves all configuration and state data on the device. The query parameter content=config is used at the end of the resource URI to limit the retrieved data to configuration data only in Example 14-47. (Query parameters are covered in detail later in this chapter.)

Example 14-47 *GET Request to the Datastore Resource*

```
! GET request to the Datastore Resource with parameter content=config

GET /restconf/data?content=config HTTP/1.1

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

Connection: keep-alive


! The contents of the Datastore Resource limited to config data only

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

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

        <version>16.11</version>

        <boot-start-marker/>

        <boot-end-marker/>

        <banner>

            <motd>
```

```
                <banner>^C</banner>
            </motd>
        </banner>
        <memory>
            <free>
                <low-watermark>
                    <processor>80557</processor>
                </low-watermark>
            </free>
        </memory>
--------- output truncated for brevity ---------
```

The Schema Resource

Unlike NETCONF, RESTCONF is not session based; therefore, there is no capability exchange between client and server. The path segment /ietf-yang-library:module-state is defined for discovering what YANG modules are supported by the server. This resource is a child resource to the datastore resource. If you send a GET request to the URI https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data/ietf-yang-library:modules-state/, you get a list of supported YANG modules on the IOS XE sandbox, as shown in Example 14-48.

Example 14-48 *GET Request to the /ietf-yang-library:modules-state/ Segment to Identify the Supported YANG Modules on the Router*

```
! GET request to retrieve the list of supported YANG modules
GET /restconf/data/ietf-yang-library:modules-state/ HTTP/1.1
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
Connection: keep-alive

! Response message listing all supported modules in the message body
HTTP/1.1 200 OK
Server: nginx
Date: Sat, 07 Dec 2019 13:08:17 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

<modules-state xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library"
xmlns:yanglib="urn:ietf:params:xml:ns:yang:ietf-yang-library">
    <module-set-id>cede0ab9198ec38357734bdd25b13778</module-set-id>
--------- output omitted for brevity ---------
    <module>
        <name>ietf-restconf</name>
        <revision>2017-01-26</revision>
        <schema>https://10.10.20.48:443/restconf/tailf/modules/ietf-restconf/2017-01-
26</schema>
        <namespace>urn:ietf:params:xml:ns:yang:ietf-restconf</namespace>
        <conformance-type>implement</conformance-type>
    </module>
    <module>
        <name>ietf-restconf-monitoring</name>
        <revision>2017-01-26</revision>
        <schema>https://10.10.20.48:443/restconf/tailf/modules/ietf-restconf-
monitoring/2017-01-26</schema>
```

```
        <namespace>urn:ietf:params:xml:ns:yang:ietf-restconf-monitoring</namespace>
        <conformance-type>implement</conformance-type>
    </module>
<module>
        <name>ietf-routing</name>
        <revision>2015-05-25</revision>
        <schema>https://10.10.20.48:443/restconf/tailf/modules/ietf-routing/2015-05-
25</schema>
        <namespace>urn:ietf:params:xml:ns:yang:ietf-routing</namespace>
        <feature>multiple-ribs</feature>
        <feature>router-id</feature>
        <deviation>
            <name>cisco-xe-ietf-routing-deviation</name>
            <revision>2016-07-09</revision>
        </deviation>
        <conformance-type>implement</conformance-type>
    </module>

--------- output omitted for brevity ---------

</modules-state>
```

The schema resource is a resource that identifies the schema of a specific YANG module in the list in Example 14-48. Each YANG module has a schema resource defined as a child resource. Therefore, to retrieve the schema of, say, the ietf-routing YANG module listed in Example 14-48, you perform a GET request to the URI identified in the <schema> XML node. Keep in mind that you need to replace the https://10.10.20.48:443/ path segment in Example 14-48 with your device's *public* details (10.10.20.48 is the IP address configured on the management interface on the router and is not reachable over the Internet since it is a private IP address). In this case, in order to retrieve the schema resource for the ietf-routing YANG module on the Cisco IOS XE sandbox, you need to perform a GET request to https://ios-xe-mgmt-latest.cisco.com:9443/restconf/tailf/modules/ietf-routing/2015-05-25, as shown in Example 14-49. For the sake of brevity, the request is omitted in the example, and only the response is shown.

Example 14-49 *GET Request to the Schema Resource of the ietf-routing YANG Module*

```
! Response message showing the schema of the ietf-routing module
HTTP/1.1 200 OK
Server: nginx
Date: Sat, 07 Dec 2019 13:28:20 GMT
Content-Type: application/yang
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Pragma: no-cache

   module ietf-routing {
     namespace "urn:ietf:params:xml:ns:yang:ietf-routing";
     prefix "rt";
     import ietf-yang-types {
       prefix "yang";
     }
     import ietf-interfaces {
       prefix "if";
     }
     organization
       "IETF NETMOD (NETCONF Data Modeling Language) Working Group";
     contact
       "WG Web:   <http://tools.ietf.org/wg/netmod/>
        WG List:  <mailto:netmod@ietf.org>
        WG Chair: Thomas Nadeau
                  <mailto:tnadeau@lucidvision.com>
```

```
        WG Chair: Juergen Schoenwaelder

                  <mailto:j.schoenwaelder@jacobs-university.de>

        Editor:   Ladislav Lhotka

                  <mailto:lhotka@nic.cz>";

     description

       "This YANG module defines essential components for the management

        of a routing subsystem.


--------- output truncated for brevity ---------
```

The Data Resource

Not to be confused with the datastore resource, which is identified by the /data path segment, a *data resource* is any piece of configuration or state data on a device that is managed as a single component. In other words, a data resource can be *uniquely* targeted using the URI in the HTTP request. A data resource maps to a data node in a YANG module—typically a leaf, leaf-list, container, or list.

A data resource is identified using a URI path segment that depends on its YANG data type.

A container is identified using the path segment */{yang_module_name}:{container_name}*. We can add this to the URI segments covered so far so that the full URI for a container is **https://**{*device_address*}**:**{*port*}/{*api_resource*}**/data/**{*yang_module_name*}**:**{*container_name*}.

The YANG module ietf-interfaces has two top-level containers: interfaces and interfaces-state. The first represents the interface-configurable parameters, and the second represents the read-only interfaces state data and statistics. Each of them is a resource and can be accessed on the IOS XE sandbox via the URIs https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data/ietf-interfaces:interfaces and https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data/ietf-interfaces:interfaces-state. Example 14-50 shows a GET request message and the resulting response for the interfaces container.

Example 14-50 *GET Request to the interfaces Container in the ietf-interfaces YANG Module*

```
! Request message to the interfaces container

GET /restconf/data/ietf-interfaces:interfaces HTTP/1.1

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

Connection: keep-alive


! Response message displaying the interfaces container

HTTP/1.1 200 OK

Server: nginx

Date: Sun, 08 Dec 2019 04:07:35 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>

        <description>MANAGEMENT INTERFACE - DON'T TOUCH ME</description>

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

type">ianaift:ethernetCsmacd</type>

        <enabled>true</enabled>

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

            <address>

                <ip>10.10.20.48</ip>

                <netmask>255.255.255.0</netmask>

            </address>
```

```
        </ipv4>
        <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
    </ipv6>
    </interface>
    <interface>
        <name>GigabitEthernet2</name>
        <description>DO NOT TOUCH ME</description>
        <type xmlns:ianaift="urn:ietf:params:xml:ns:yang:iana-if-
type">ianaift:ethernetCsmacd</type>
        <enabled>true</enabled>
        <ipv4 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
            <address>
                <ip>10.255.255.1</ip>
                <netmask>255.255.255.0</netmask>
            </address>
        </ipv4>
        <ipv6 xmlns="urn:ietf:params:xml:ns:yang:ietf-ip">
    </ipv6>
    </interface>
    <interface>


--------- output omitted for brevity ---------


</interfaces>
```

YANG lists and leaf lists are addressed similarly. A list is addressed using the path segment */{yang_module_name}:{container_name}/{list_name}={key-value-1}*[,..][,*key-value-n*], and a leaf list is addressed using the path segment */{yang_module_name}:{container_name}/{leaf_list_name}={value}*. Recall from Chapter 13 that a list contains one or more instances, and each instance is differentiated from the others by one or more keys. On the other hand, a leaf list is a list of leafs, each having a single value.

In the ietf-interfaces YANG module, the <interfaces> element represents a container, under which resides the element <interface> that represents a list. To retrieve only the list instance for interface GigabitEthernet3, the URI https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet3 is used. Note that the list key is the leaf represented by the element <name>. Example 14-51 shows the result.

Example 14-51 *Response Message to a GET Request to the List Instance for Interface GigabitEthernet3*

```
! Response message displaying the list instance for interface GigabitEthernet3
HTTP/1.1 200 OK
Server: nginx
Date: Sun, 08 Dec 2019 04:39:20 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>
    <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>10.10.30.1</ip>
            <netmask>255.255.255.0</netmask>
        </address>
    </ipv4>
```

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

Finally, a leaf is addressed using the path segment */{yang_module_name}:{container_name}/{full-path-to-leaf}/{leaf_name}*. To target a leaf, you include the full path to the leaf in the URI. For example, the <description> element is a leaf. Therefore, to retrieve the interface description of interface GigabitEthernet3 on the IOS XE sandbox, you use the URI https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet3/description, as shown in Example 14-52.

Example 14-52 *Retrieving the Value of the Leaf element <description> for Interface GigabitEthernet3*

```
! Request message for leaf description for interface GigabitEthernet3
GET /restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet3/description
HTTP/1.1
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
Connection: keep-alive

! Response message displaying the list instance for interface GigabitEthernet3
HTTP/1.1 200 OK
Server: nginx
Date: Sun, 08 Dec 2019 04:59:36 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

<description xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"
xmlns:if="urn:ietf:params:xml:ns:yang:ietf-interfaces">
  Network Interface
</description>
```

The Operations Resource

Recall from Chapter 13 that you can extend the set of NETCONF RPCs by defining new RPC operations in a YANG module by using the *rpc* statement. You can also define an operation that may be performed on a specific container or list node by using the *action* statement. Both of these constructs can be addressed in RESTCONF by using an operations resource, which is the second type of resource under the API resource (refer to Figure 14-8).

You can retrieve the list of operations defined using an rpc statement in all modules supported by the device by sending a GET request to the /operations path segment. In the case of the IOS XE sandbox, the full URI is https://ios-xe-mgmt-latest.cisco.com:9443/restconf/operations. Example 14-53 shows the response to the GET request to that URI.

Example 14-53 *Response to a GET Request to the Operations Resource*

```
! The contents of the operations resource
HTTP/1.1 200 OK
Server: nginx
Date: Mon, 25 Nov 2019 07:46:24 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

<operations xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf">
    <ios-xe-rpc:switch xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
```

```
rpc">/restconf/operations/Cisco-IOS-XE-rpc:switch</ios-xe-rpc:switch>
    <ios-xe-rpc:default xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:default</ios-xe-rpc:default>
    <ios-xe-rpc:clear xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:clear</ios-xe-rpc:clear>
    <ios-xe-rpc:release xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:release</ios-xe-rpc:release>
    <ios-xe-rpc:reload xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:reload</ios-xe-rpc:reload>
    <ios-xe-rpc:cellular xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:cellular</ios-xe-rpc:cellular>
    <ios-xe-rpc:license xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:license</ios-xe-rpc:license>
    <ios-xe-rpc:service xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:service</ios-xe-rpc:service>
    <ios-xe-rpc:virtual-service xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:virtual-service</ios-xe-rpc:virtual-service>
    <ios-xe-rpc:copy xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:copy</ios-xe-rpc:copy>
    <ios-xe-rpc:delete xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:delete</ios-xe-rpc:delete>
    <ios-xe-rpc:app-hosting xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:app-hosting</ios-xe-rpc:app-hosting>
    <ios-xe-rpc:guestshell xmlns:ios-xe-rpc="http://cisco.com/ns/yang/Cisco-IOS-XE-
rpc">/restconf/operations/Cisco-IOS-XE-rpc:guestshell</ios-xe-rpc:guestshell>

--------- output omitted for brevity ---------

</operations>
```

The response contains a number of child resources, each defining a different data model–specific RPC operation. To invoke a particular operation, a POST request is sent to the URI of the operation, formatted as **https://**{*device_address*}**:**{*port*}/{*api_resource*}**/operations/**{*yang_module_name*}**:**{*operation*}. For example, to reload the IOS XE sandbox, you perform a POST to the URI https://ios-xe-mgmt-latest.cisco.com:9443/restconf/operations/Cisco-IOS-XE-rpc:reload.

Note

Do not try sending a POST request to the URI of the reload operation to a device in a production network unless you intend to reload the device.

An operations resource of the second type—the one defined using a YANG action statement—is defined for a specific node in the YANG module; therefore, it is addressed using the URI **https://**{*device_address*}**:**{*port*}/{*api_resource*}**/data/**{*path-to-data-resource*}/{*action*}. The available actions for any data resource cannot be discovered. Each of these actions is defined in the corresponding YANG module.

The YANG Library Version Resource

The final second-level resource under the API resource is the yang-library-version resource, which, as the name implies, indicates the version of the YANG library supported by the device. This resource is identified by the path segment /yang-library-version. Example 14-54 shows the response message to a GET request to the URI https://ios-xe-mgmt-latest.cisco.com:9443/restconf/yang-library-version.

Example 14-54 *GET Request to Retrieve the YANG Library Version Resource*

```
! The response message showing the YANG library version
HTTP/1.1 200 OK
Server: nginx
Date: Wed, 11 Dec 2019 20:27:36 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
```

```
<yang-library-version xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library">2016-06-
21</yang-library-version>
```

The message body in the response shows that the YANG library version on this device is 2016-06-21.

Methods and the RESTCONF Operations Layer

Now that you know how to construct a URI to target a specific resource, in this section you will see how to use HTTP methods to act on these resources. Methods can be roughly classified into two categories: methods to retrieve resource representations and methods to edit resources. Because RESTCONF is a RESTful protocol, much of the material in this section will sound very familiar if you have already studied Chapters 7 and 8.

Retrieving Data: OPTIONS, GET, and HEAD

You use the OPTIONS method in requests to discover which methods are supported for a specific resource. The allowed methods are returned by the server in the Allow header field of the response.

Take, for example, the two containers interfaces and interfaces-state, both defined in the YANG module ietf-interfaces. The first holds the interface configuration, and the second holds the interface (non-configurable) state data. Example 14-55 shows an OPTIONS request being sent to each resource on the IOS XE sandbox.

Example 14-55 *Using the OPTIONS Method to Discover the Allowed Methods for the interfaces and interfaces-state Resources*

```
! OPTIONS request and response for the interfaces Resource
OPTIONS /restconf/data/ietf-interfaces:interfaces/ HTTP/1.1
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/1.1 200 OK
Server: nginx
Date: Fri, 13 Dec 2019 21:32:57 GMT
Content-Type: text/html
Content-Length: 0
Connection: keep-alive
Allow: DELETE, GET, HEAD, PATCH, POST, PUT, OPTIONS
Cache-Control: private, no-cache, must-revalidate, proxy-revalidate
Accept-Patch: application/yang-data+xml, application/yang-data+json
Pragma: no-cache

! OPTIONS request and response for the interfaces-state Resource
OPTIONS /restconf/data/ietf-interfaces:interfaces-state/ HTTP/1.1
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/1.1 200 OK
Server: nginx
Date: Fri, 13 Dec 2019 21:33:04 GMT
Content-Type: text/html
Content-Length: 0
Connection: keep-alive
```

```
Allow: GET, HEAD, OPTIONS

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

Pragma: no-cache
```

The Allow header field in the first response in Example 14-55 shows that the resource accepts requests that use the DELETE, GET, HEAD, PATCH, POST, PUT, and OPTIONS methods. The second response message shows that the resource, being read-only, accepts requests that use the GET, HEAD, and OPTIONS methods only.

You have seen how GET works in other examples in this chapter. Now you will see how GET fails ! If a GET request is made for a resource to which the client does not have read access, the server responds with either a "401 Unauthorized" or a "404 Not Found" message. If a GET request is made for a resource that does not exist, or for an instance of a YANG list or leaf list that does not exist, the server responds with a "404 Not Found" message.

The GET method is supported for all resource types except operations resources (that are defined in YANG modules using the rpc and action statements). If a GET request is made for an operations resource, the server responds with a "405 Method Not Allowed" message.

A GET request may be sent for a specific instance of a list or leaf list, such as a GET, to the URI https://ios-xe-mgmt-latest.cisco.com:9443/restconf/data/ietf-interfaces:interfaces/interface=GigabitEthernet1, where GigbaitEthernet1 is an instance of the list interface. What if a GET request does not specify a particular instance of the list interface ? If the response returned from the server is in XML, that would not be allowed, and the response status line would be "400 Bad Request," indicating an error condition. If the response is in JSON, the response would be a normal 200 OK response, and the response body would be composed of a JSON array listing all instances of the list or leaf list.

As you saw in Chapter 7, a client indicates the preferred encoding for the response message body by specifying this encoding in the request Accept header. Building on that, in the first part of example 14-56, the Accept header in the GET request for the interface list specifies XML as the preferred encoding for the response. As a result, the response message has a "400 Bad Request" status line. Alternatively, when the Accept header specifies JSON as the requested encoding in the second part of the example, the response message body correctly lists all interfaces in a JSON array.

Example 14-56 *Sending a GET Request for More Than One List Instance with the Accept Header Set to XML and then to JSON*

```
! GET request for the interface list with Accept header = application/yang-data+xml

GET /restconf/data/ietf-interfaces:interfaces/interface HTTP/1.1

Authorization: Basic ZGV2ZWxvcGVyOkMxc2NvMTIzNDU=

User-Agent: PostmanRuntime/7.20.1

Accept: application/yang-data+xml

Cache-Control: no-cache

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

Accept-Encoding: gzip, deflate

Connection: keep-alive


HTTP/1.1 400 Bad Request

Server: nginx

Date: Fri, 13 Dec 2019 21:51:36 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>too many instances: 3</error-message>

        <error-tag>invalid-value</error-tag>

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

    </error>

</errors>


! GET request for the interface list with Accept header = application/yang-data+json

GET /restconf/data/ietf-interfaces:interfaces/interface HTTP/1.1

Accept: application/yang-data+json

Authorization: Basic ZGV2ZWxvcGVyOkMxc2NvMTIzNDU=

User-Agent: PostmanRuntime/7.20.1

Cache-Control: no-cache

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