CCNA Exam Prep: Back to Networking Basics with Hank Preston and Patrick Gargano

GET a Head(er) of REST APIs

It is time to dive into network automation skills needed to succeed in the journey towards your CCNA certification. Learn everything you need to describe authentication, decode any possible CRUD operation, and handle JSON and XML results when working with REST APIs in this bonus CCNA Prep webinar.

In this lab you will:

- Explore each of the basic "CRUD" operations using RESTCONF and an IOS XE router
- Learn how to use Basic authentication with REST APIs
- Leverage Headers such as `Accept` and `Content-type` to influence data encoding formats
- Explore returned data and details from REST API servers

Setup and Scenario

This set of lab based demonstrations includes a very simple network composed of 1 Linux desktop and 1 Cisco 8000v router running IOS XE. The router has been pre-configured to enable the RESTCONF protocol. You will use the Linux desktop to send RESTCONF requests to the router using `curl`, a basic command line web client available on most common operating systems including Windows, Mac and Linux.

> NOTE: RESTCONF is an IETF standard REST API for networking. It is related to the standards NETCONF and YANG.
>
> - RESTCONF - RFC 8040
> - NETCONF - RFC 6241
> - YANG 1.1 - RFC 7950

*Be sure to **START** the lab before continuing to the demo labs.*

Making our first REST API Call

Before we can use RESTCONF to make configuration changes, or even lookup interesting bits of information, we need to successfully connect to the router. This will involve crafting a REST API request using the correct address, authentication, and URL format.

1. Open up the console for `desktop` and login.
2. We will start with a simple "hello world" kind of request, ask the router for it's hostname. The RESTCONF URL for an IOS XE hostname is:

   ```
   https://{router-address}/restconf/data/Cisco-IOS-XE-native:native/hostname
   ```

   `{router-address}` can be the IP address or FQDN for the router's management address where RESTCONF is available.
3. Use `curl` to send a request to `cat8000v`.

   ```
   curl https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/hostname
   ```

   ▼ Expected Output

   ```
   curl: (60) SSL certificate problem: self signed certificate
    More details here: https://curl.se/docs/sslcerts.html

    curl failed to verify the legitimacy of the server and therefore could not
    establish a secure connection to it. To learn more about this situation and
    how to fix it, please visit the web page mentioned above.
   ```
4. RESTCONF is a "secure" protocol and runs over HTTPS using SSL/TLS for security. However, by default the certificate used for TLS is "self-signed" and not immediately "trusted" by your client. This is similar to webbased network management applications that require you to "Accept the risk" in order to access the interface.
5. We will "accept the risk" by adding the argument `--insecure` (or the shorter argument `-k`) to the request.

   ```
   curl --insecure https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/hostname
   ```

   ▼ Expected Output

   ```
   <errors xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf">
     <error>
       <error-type>protocol</error-type>
       <error-tag>access-denied</error-tag>
     </error>
   </errors>
   ```
6. We get an `access-denied` error because we have failed to provide credentials to authenticate to the router. Just because we are using an API, doesn't mean we don't need to properly authenticate. There are different options for authentication with RESTAPIs, the simplest is called "Basic Authentication" and uses a username and password to authenticate. RESTCONF on IOS XE uses "basic" authentication.
7. Provide the username and password to the `curl` request with the `--user username:password` argument. (Or the shorter `-u username:password` argument).

```
curl --insecure --user cisco:cisco \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/hostname
```

The command is getting longer, so we will use the `\` character to provide "multi-line" commands in the bash terminal.

▼ Expected Output

```
<hostname xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native"  xmlns:ios="http://cisco.com/ns/yang/Cisco-IOS-
```

Great job!!! You've sent your first successful REST API request to your router. Continue to peak behind the curtain to see more about how this works.

Peaking behind the curtain

Looking at Headers and Status Codes with the "Verbose" option

We sent an **HTTP Request** to the router, and it replied with an **HTTP RESPONSE**. The *body* of the response was displayed by `curl` and contained an XML encoded string that included the current `hostname` of the router.

```
<hostname xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native"  xmlns:ios="http://cisco.com/ns/yang/Cisco-IOS-XE-na
```

1. In addition to the "body" (or data) returned in a response, there is a lot more that is not displayed by default by `curl`. We can see the full response by adding the argument `--verbose` (or `-v`) to the request.

```
curl --insecure --verbose --user cisco:cisco \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/hostname
```

▼ Expected Output

```
*   Trying 10.1.1.254:443...
 * Connected to 10.1.1.254 (10.1.1.254) port 443 (#0)
 * ALPN: offers h2,http/1.1
 * TLSv1.3 (OUT), TLS handshake, Client hello (1):
 * TLSv1.3 (IN), TLS handshake, Server hello (2):
 * TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
 * TLSv1.3 (OUT), TLS handshake, Client hello (1):
 * TLSv1.3 (IN), TLS handshake, Server hello (2):
 * TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
 * TLSv1.3 (IN), TLS handshake, Certificate (11):
 * TLSv1.3 (IN), TLS handshake, CERT verify (15):
 * TLSv1.3 (IN), TLS handshake, Finished (20):
 * TLSv1.3 (OUT), TLS handshake, Finished (20):
 * SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
 * ALPN: server accepted http/1.1
 * Server certificate:
 *  subject: CN=IOS-Self-Signed-Certificate-1736198370
 *  start date: Nov 25 00:17:20 2024 GMT
 *  expire date: Nov 25 00:17:20 2034 GMT
 *  issuer: CN=IOS-Self-Signed-Certificate-1736198370
 *  SSL certificate verify result: self signed certificate (18), continuing anyway.
 * using HTTP/1.1
 * Server auth using Basic with user 'cisco'
 > GET /restconf/data/Cisco-IOS-XE-native:native/hostname HTTP/1.1
 > Host: 10.1.1.254
 > Authorization: Basic Y2lzY286Y2lzY28=
 > User-Agent: curl/8.0.1
 > Accept: */*
 >
 * TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
 * TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
 * old SSL session ID is stale, removing
 < HTTP/1.1 200 OK
 < Server: openresty
 < Date: Mon, 25 Nov 2024 00:44:43 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
 < Content-Security-Policy: default-src 'self'; block-all-mixed-content; base-uri 'self'; frame-ancestors 'none
 < Strict-Transport-Security: max-age=15552000; includeSubDomains
 < X-Content-Type-Options: nosniff
 < X-Frame-Options: DENY
 < X-XSS-Protection: 1; mode=block
 <

 <hostname xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native"  xmlns:ios="http://cisco.com/ns/yang/Cisco-IOS-
```

Each line of the "verbose output" starts with a character that indicates what the line represents.

- `*` - These lines represent basic connection details between client and server. Most of these lines relate to the TLS negotiation.

    - `>` - These lines represent the HTTP **REQUEST** that `curl` crafted and sent to the server.
    - `<` - These lines represent the HTTP **RESPONSE** that was returned by the web server.

2. Look at the first RESPONSE line. It should be `< HTTP/1.1 200 OK`. The first part of the line indicates the HTTP version in use. Nearly all web servers run `HTTP/1.1` today. The more interesting part is the second half of the line - `200 OK`. This represents the HTTP "status code". Status codes include a number, `200`, and a message, `OK`.

3. There are dozens of possible status codes, but there are only a handful that a CCNA needs to worry about.

    - Status code categories
        - `2xx successful` - Any code in the `200s` can be considered positive. Whatever was expected by the request likely happened.
        - `4xx client error` - Any code in the `400s` indicates something was wrong with the REQUEST. A `404 Not Found` is an example where the REQUESTed data was not available on the device. Perhaps you tried looking up an interface that doesn't exist?
        - `5xx server error` - Any code in the `500s` indicates something is wrong with the server. The server could be overloaded, or down and broken.
    - Status codes worth remembering
        - `200 OK` - Most common "good code"
        - `401 Unauthorized` - Indicates failed or missing authentication
        - `404 Not Found` - Indicates the requested resource doesn't exist

4. Return to the RESPONSE lines, and find the line `< Content-Type: application/yang-data+xml`. This is one of the many "headers" included in the response. "Headers" are a way for both request and response objects to provide details about the request being sent or reponse being returned. The `Content-type` header indicates the data format of the "body", so that clients and/or servers know how to "read" the data.

5. In this case, the server returned the hostname in the format of `application/yang-data+xml`. The key part being the `xml` portion. Which matches because we did infact get `xml` data back.

```
<hostname xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native"  xmlns:ios="http://cisco.com/ns/yang/Cisco-IOS-
```

6. XML is the "default" content type used by the router, but we can also get JSON data back instead. We'll do that in the next part.

7. Now look at the REQUEST lines, and find the `> Authorization: Basic Y2lzY286Y2lzY28=` header. This is where the `username:password` is included in the REQUEST to provide authentication to the server. We'll look at this part more in a bit.

Requesting JSON data from the server

XML data is fine, but many developers and engineers find JSON data easier to work with. Luckily, RESTCONF supports JSON as well as XML. Let's ask the router to send the hostname to us in JSON format.

1. We can request JSON data by including a header `Accept: application/yang-data+json` in our request. Headers are added to `curl` with the `--header` (or `-H`) option.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/hostname
```

▼ Expected Output

```
{
    "Cisco-IOS-XE-native:hostname": "cat8000v"
}
```

Do more! Try sending a `verbose` request and compare the REQUEST and RESPONSE headers to see the differences.

The choice of XML or JSON for data formats is up to the engineer's preference. Do you prefer XML or JSON?

Understanding BASIC Authentication

Let's look more at the authentication that is used by the REST API. Basic authentication is very easy to leverage, and if you look back at the REQUEST header `Authorization: Basic Y2lzY286Y2lzY28=`, it might even look like it is safe and secure. Well, let's see how "secure" it is.

1. Start by resending verbose request for the hostname.

```
curl --insecure --verbose --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/hostname
```

2. Locate the `Authorization` header.

```
> Authorization: Basic Y2lzY286Y2lzY28=
```

3. `Authorization` headers will indicate the type of authentication (or authorization) in use, followed by a string that represents the credential the server will use to verify access to the requested data.

4. "Basic" authentication creates the credential by performing a base64 encoding of the string `username:password` (where the actual username and password are used.)
5. "Encoding" is NOT "encryption". And an "encoded" string, can be "decoded". The server will do this when it recieves the request.

6. But we can decode it ourselves using the `base64` program.

```
echo Y2lzY286Y2lzY28= | base64 -d
```

▼ Expected Output

```
cisco:ciscodesktop:~$
```

- Look at that, the username and password are easily readable!
- *The encoded string does NOT include a "new line character" which is why the terminal prompt shows up on the same line as the decoded credential.*

7. The `curl` option `--user` does the encoding of the credentials and addition of the `Authorization` header for us. But we can also do this ourselves.

8. While we already know what `cisco:cisco` encoded with base64 is, we can generate this with the `base64` program.

```
echo -n cisco:cisco | base64
```

> Note: By default `echo` will add a "new line" character in addition to the string provided. This will cause problems for authorization. The `-n` argument does NOT add the new-line character.

▼ Expected Output

```
Y2lzY286Y2lzY28=
```

9. Now remove the `--user cisco:cisco` argument from `curl` and replace it with the explicit `Authorization` header.

```
curl --insecure \
  --header "Authorization: Basic Y2lzY286Y2lzY28=" \
  --header "Accept: application/yang-data+json" \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/hostname
```

- You should get the hostname in JSON format returned.
- Feel free to try again with the `--verbose` option to check the headers.

Now that we've seen how Basic authentication works, we can go back to using `--user` to save us the step of encoding on our own. But never forget that basic authentication is NOT a secure method of transporting credentials. Thankfully, the request and response headers are ENCRYPTED with TLS because RESTCONF runs over HTTPs and not HTTP.

Let's get our CRUD on!

The time has come to see how we can really interact with a device configuration using RESTCONF. REST API actions are often called "CRUD" operations. This acronym stands for **Create**, **R**ead, **U**pdate, and **D**elete. Each of these actions are associated with a related **HTTP Method**.

Action	HTTP Method	Description
Create	POST	Used to create a new object/resource
Read	GET	Retrieve details about an object/resource
Update	PUT	Replace or update a resource/object
Delete	DELETE	Remove a resource/object

We will explore each of these operations by exploring interfaces, and loopback interfaces, on our router.

The base URL for the IOS XE interface list is:

```
https://{device address}/restconf/data/Cisco-IOS-XE-native:native/interface
```

"READ" the router interfaces with `GET`

1. `curl` assumes you are making a `GET` request with a basic command. But we will be explicit by adding the `--request` (or `-X`) argument.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --request GET \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface
```

▼ Expected Output


```
{
"Cisco-IOS-XE-native:interface": {
    "GigabitEthernet": [
    {
        "name": "1",
        "description": "Link to desktop",
        "ip": {
        "address": {
            "primary": {
            "address": "10.1.1.254",
            "mask": "255.255.255.0"
            }
        }
        },
        "logging": {
        "event": {
            "link-status": [null]
        }
```

```
        },
        "access-session": {
        "host-mode": "multi-auth"
        },
        "Cisco-IOS-XE-ethernet:negotiation": {
        "auto": true
        }
    },
    {
        "name": "2",
        "shutdown": [null],
        "logging": {
        "event": {
            "link-status": [null]
        }
        },
        "access-session": {
        "host-mode": "multi-auth"
        },
        "Cisco-IOS-XE-ethernet:negotiation": {
        "auto": true
        }
    },
    {
        "name": "3",
        "shutdown": [null],
        "logging": {
        "event": {
            "link-status": [null]
        }
        },
        "access-session": {
        "host-mode": "multi-auth"
        },
        "Cisco-IOS-XE-ethernet:negotiation": {
        "auto": true
        }
    },
    {
        "name": "4",
        "shutdown": [null],
        "logging": {
        "event": {
            "link-status": [null]
        }
        },
        "access-session": {
        "host-mode": "multi-auth"
        },
        "Cisco-IOS-XE-ethernet:negotiation": {
        "auto": true
        }
    }
    ],
    "Loopback": [
    {
        "name": 1,
        "description": "Created in Day0 Config",
        "ip": {
        "address": {
            "primary": {
            "address": "192.168.1.1",
            "mask": "255.255.255.255"
            }
        }
        },
        "logging": {
        "event": {
            "link-status": [null]
        }
        }
    },
    {
        "name": 2,
        "description": "Created in Day0 Config",
        "ip": {
        "address": {
            "primary": {
            "address": "192.168.2.2",
            "mask": "255.255.255.255"
            }
```

```
        }
        },
        "logging": {
        "event": {
            "link-status": [null]
        }
        }
    }
    ]
}
}
```

2. Let's limit the results to just the `Loopback` interfaces.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --request GET \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback
```

3. We can specify just `Loopback1` in the URL with `Loopback=1`.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --request GET \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=1
```

4. Perhaps we only want the IP address information for the interface.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --request GET \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=1/ip/address
```

▼ Expected Output

```
{
  "Cisco-IOS-XE-native:address": {
      "primary": {
          "address": "192.168.1.1",
          "mask": "255.255.255.255"
      }
  }
}
```

"CREATING" a new Loopback with `POST`

Now that we've seen how to "READ" details about Loopback interfaces, we'll learn how to create a new one. The key details about performing a POST to CREATE and object are:

- We will be ***sending*** data to the server along with our REQUEST
- We need to specify the `Content-type` of the data we are sending, just like the server specifies in the RESPONSE
- The data format for a new Loopback looks just like the returned data when we GET details about a Loopback
- We need to POST the data to the appropriate URL/resource. For RESTCONF, we'll send our POST to the `Cisco-IOS-XE-native:native/interface` "list"
- We want to create the equivalent of this configuration using RESTCONF.

```
interface Loopback100
  description Created with RESTCONF
  ip address 192.168.100.100 255.255.255.255
end
```

- Use the output for `Loopback=1` to craft a JSON body with the new values for `Loopback100`.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --request GET \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=1
```

▼ Loopback 100 JSON Body

```
{
"Cisco-IOS-XE-native:Loopback": [
    {
    "name": 100,
    "description": "Created with RESTCONF",
    "ip": {
        "address": {
        "primary": {
            "address": "192.168.100.100",
```

```
            "mask": "255.255.255.255"
        }
        }
    },
    "logging": {
        "event": {
        "link-status": [null]
        }
    }
    }
]
}
```

1. The JSON body is included with the request using the `--data` argument.
    - Surround the JSON body in single quotes (`'`). This will allow the JSON body to span across multiple lines in the terminal and not interfere with the double quotes (`"`) that are part of the JSON body itself.
    - Include a `Content-type: application/yang-data+json` header
    - Include the `Accept` header as well so tell the server to return any data in JSON format

    ```
    curl --insecure --user cisco:cisco \
    --header "Accept: application/yang-data+json" \
    --header "Content-type: application/yang-data+json" \
    --request POST \
    --data '
      {
      "Cisco-IOS-XE-native:Loopback": [
          {
          "name": 100,
          "description": "Created with RESTCONF",
          "ip": {
              "address": {
              "primary": {
                  "address": "192.168.100.100",
                  "mask": "255.255.255.255"
              }
              }
          },
          "logging": {
              "event": {
              "link-status": [null]
              }
          }
          }
      ]
      }
    ' \
    https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface
    ```

    Note: If everything goes correctly, there will be NO data returned from the server

2. Check to see what happens if you resend the request a second time.

▼ Expected Output

```
{
"ietf-restconf:errors": {
    "error": [
    {
        "error-type": "application",
        "error-tag": "data-exists",
        "error-path": "/Cisco-IOS-XE-native:native/interface",
        "error-message": "object already exists: /ios:native/ios:interface/ios:Loopback[ios:name='100']"
    }
    ]
}
}
```

- The error is pretty straightforward. We can't create `Loopback100` because it already exists. Looks like it DID work!

3. Now send a GET request for `Loopback=100` to verify the interface was indeed created.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --request GET \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100
```

- Did you get a result? Does it match what you created above?

"UPDATING" interface details with `PUT`

You can use `PUT` to replace data with new data. Let's see how we can change the description for our interface.

1. Start by getting the current interface description.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --request GET \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100/description
```

▼ Expected Output

```
{
    "Cisco-IOS-XE-native:description": "Created with RESTCONF"
}
```

2. A `PUT` request is very similar to `POST` request. Use the returned data from the above request to craft the new `data` to send. You will send the request to the URL for the current description. Effectively `PUTTING` a new description at that address.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --header "Content-type: application/yang-data+json" \
  --request PUT \
  --data '
    {
        "Cisco-IOS-XE-native:description": "UPDATED with RESTCONF"
    }
  ' \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100/description
```

3. Send the `GET` request to check the results.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --request GET \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100/description
```

   - Did the interface description change?

You can use this same approach to update other parts of the interface configuration. Try to change the IP address for `Loopback100` to `10.100.100.100`.

> The address for the `Loopback100` ip is: `https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100/ip`

▼ Answer: PUTTING a new IP address

```
curl --insecure --user cisco:cisco \
    --header "Accept: application/yang-data+json" \
    --header "Content-type: application/yang-data+json" \
    --request PUT \
    --data '
        {
        "Cisco-IOS-XE-native:ip": {
            "address": {
            "primary": {
                "address": "10.100.100.100",
                "mask": "255.255.255.255"
            }
            }
        }
        }
    ' \
    https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100/ip
```

Deleting an interface

The last of the CRUD actions is **DELETE**. And next to `GET`, the `DELETE` operation is the simplest to use.

1. Start by `GETing` `Loopback100` to verify it is still there.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --request GET \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100
```

2. Now simply change from `GET` to `DELETE` and resend the request. There is no data required to delete a resource, you just target an address with the action.

```
curl --insecure --user cisco:cisco \
  --header "Accept: application/yang-data+json" \
  --request DELETE \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100
```

- Just like when we created the interface with `POST`, a successful `DELETE` does NOT return any data.

3. Try to `DELETE` it again, but this time add the `--verbose` argument to the command.

```
curl --insecure --user cisco:cisco --verbose \
  --header "Accept: application/yang-data+json" \
  --request DELETE \
  https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100
```

   - Find the first RESPONSE line. What status code was returned? Why was this code returned?

4. Now try to `GET Loopback100` again, also with the `--verbose` flag.

   - Before you send it off, guess what the results will be? What status code will be returned?

     ```
     curl --insecure --user cisco:cisco --verbose \
     --header "Accept: application/yang-data+json" \
     --request GET \
     https://10.1.1.254/restconf/data/Cisco-IOS-XE-native:native/interface/Loopback=100
     ```

   - Were you right?