# ArubaOS-CX REST API Guide for 10.02


a Hewlett Packard
Enterprise company

Part Number: 5200-5330a
Published: January 2019
Edition: 2

## Notices

The information contained herein is subject to change without notice. The only warranties for Hewlett Packard Enterprise products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Hewlett Packard Enterprise shall not be liable for technical or editorial errors or omissions contained herein.

Confidential computer software. Valid license from Hewlett Packard Enterprise required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Links to third-party websites take you outside the Hewlett Packard Enterprise website. Hewlett Packard Enterprise has no control over and is not responsible for information outside the Hewlett Packard Enterprise website.

This guide describes the ArubaOS-CX REST API. It is intended for experienced users who require a programmatic interface to the switch and understand the schema and data relationships of the switch as modeled by the switch configuration database.

# Applicable products

This document applies to the following products:

- Aruba 8400 Switch Series (JL375A, JL376A)
- Aruba 8320 Switch Series (JL479A, JL579A, JL581A)
- Aruba 8325 Switch Series (JL624A, JL625A, JL626A, JL627A, JL635A, JL636A)

# About the examples

Examples in this document are representative and might not match your particular switch or environment.

The slot and port numbers in this document are for illustration only and might be unavailable on your device.

The software notation for identifying interfaces uses member/slot/port notation, such as 1/1/1. For standalone switches such as the 8320, slot is always 1.

# ArubaOS-CX REST API

Switches running the ArubaOS-CX software are fully programmable with a REST (Representational State Transfer) API, allowing easy integration with other devices both on premises and in the cloud. This programmability—combined with the Aruba Network Analytics Engine—accelerates network administrator understanding of and response to network issues.

The ArubaOS-CX REST API enables programmatic access to the ArubaOS-CX configuration and state database at the heart of the switch. By using a structured model, changes to the content and formatting of the CLI output do not affect the programs you write. And because the configuration is stored in a structured database instead of a text file, rolling back changes is easier than ever, thus dramatically reducing a risk of downtime and performance issues.

The ArubaOS-CX REST API is a web service that uses basic CRUD (Create, Read, Update, and Delete) operations performed on switch resources using HTTPS `POST`, `GET`, `PUT`, and `DELETE` methods.

A switch resource is indicated by its URI. A URI can be made up of several components, including the host name or IP address, port number, the path, and an optional query string. The ArubaOS-CX operating system includes the ArubaOS-CX REST API Reference, which is a web interface based on the Swagger UI. The ArubaOS-CX REST API Reference provides the reference documentation for the REST API, including resources, models, methods, and errors.

# ArubaOS-CX system databases

The ArubaOS-CX operating system is a modular, database-centric operating system. Every aspect of the switch configuration and state information is modeled in the ArubaOS-CX switch configuration and state database. The entire current state of the system is in the configuration database, including the following:

- Configuration information
- Status of all features
- Statistics

The ArubaOS-CX operating system also includes a time series database, which acts as a built-in network record. The time series database makes the data seamlessly available to Aruba Network Analytics Engine agents that use rules that evaluate network conditions over time. Time-series data about the resources monitored by agents are automatically collected and presented in graphs in the switch Web UI.

# Authentication of REST API sessions

When you start a REST API session, you use the POST method to access the `login` resource of the switch and pass the username and password information as data. Ensure that HTTPS is configured to use port 443. HTTPS requests to port 80 are redirected to port 443.

If the credentials are accepted, your authenticated session is started for that username, and the switch returns a cookie containing encoded session information.

In subsequent calls to the API—including to the `logout` resource—you must pass the session cookie back to the switch.

The same session cookie is shared across browser tabs and, depending on the browser, multiple browser windows. However, the same session cookie is not shared across devices and scripts. For example, if a user logs

into the Web UI from a laptop, again with a tablet, and then uses the same user name in a curl command, that user has three concurrent client sessions.

**NOTE:** The number of concurrent HTTPS sessions per client and per switch are limited. Ensure that you log out of HTTP sessions when you are finished using them.

Authentication through methods other than the session cookie, such as OAuth or certificates, is not supported. The server uses self-signed certificates.

The procedure to pass the session cookie back and forth from the switch depends on how you access the REST API.

For example:

- If you log in to the REST API using the ArubaOS-CX REST API Reference or using the Web UI and open the API Reference in another browser tab, the browser handles the session cookie for you. You do not have to save or otherwise manage the session cookie.
- If you access the REST API using another method, such as the curl tool, you must do the following:
  - Save the session cookie returned from the login request.
  - Pass that saved cookie to the switch with every subsequent request you make to the REST API.

  **IMPORTANT:** Although it is possible to pass the user name and password information as a query string in the login URL, browser logs or tools outside the switch might save the accessed URL in cleartext in log entries. Instead, Hewlett Packard Enterprise recommends that you pass the credential information as data when using programs such as curl to log in to the switch.

  In the following examples, the workstation is running a Linux-based operating system and curl version 7.35 is installed.
  - Example of logging in and obtaining the session cookie, and storing that cookie in the file `/tmp/auth_cookie` on your local workstation:

    ```
    $ curl --noproxy 192.0.2.5 -k -X POST \
    -c /tmp/auth_cookie \
    -H 'Content-Type: application/x-www-form-urlencoded' \
    "https://192.0.2.5:443/rest/v1/login" \
    --data 'username=admin&password=admin'
    ```

  - Example of passing using the `-b` curl command option to pass the cookie back to the switch:

    ```
    $ curl -k -X GET -b /tmp/auth_cookie \
    --header 'Content-Type:application/json' \
    --header 'Accept: application/json' \
    "https://192.0.2.5/rest/v1/system"
    ```

  - Example of logging out at the end of the session:

    ```
    $ curl -k -X POST -b /tmp/auth_cookie \
    --header 'Content-Type:application/json' \
    --header 'Accept: application/json' \
    "https://192.0.2.5/rest/v1/logout"
    ```

### More information

User groups and access authorization on page 9
ArubaOS-CX REST API reference summary on page 17
HTTP 404 error "Page not found" when using a write method on page 94

# User groups and access authorization

All users can log in to the switch and log out of the switch using the POST method of the `\login` and `\logout` resources. For other switch resources, the access authorization granted to a user is determined by the group to which the user belongs. Each user group is assigned number that represents a privilege level. This number is used to represent the user group in logs and in places in which the group name is too long to display.

The following user groups are supported:

| User group | Privilege level | Description |
|---|---|---|
| `operators` | 1 | Authorized for read access to non-sensitive data. |
| `administrators` | 15 | Authorized for read and write access to all switch resources. Write access also requires that the REST API is in read/write access mode. |
| `auditors` | 19 | Authorized for read access to audit log (`/logs/audit`) and event log (`/logs/event`) resources only. |

If a user attempts a request for which they are not authorized, the switch returns an HTTP 403 "Forbidden" error.

If an authorized user attempts a write request but the REST API is in read-only mode, the switch returns an HTTP 404 "Page not found" error.

# REST API access modes

The REST API supports two access modes:

- read-only (default)
- read/write

You can enable write support (POST, PUT, and DELETE methods) using the `https-server rest access-mode read-write` CLI command from the global configuration (`config`) context.

### Read-only access mode

In the read-only access mode:

- Most switch resources support GET methods only, but some resources allow PUT or POST methods to be used even when the REST access mode is read-only.

  For example, you can use POST to log into the switch, use PUT to upload a new running configuration, or use POST to upload a new firmware version.
- For most switch resources, the ArubaOS-CX REST API Reference does not show any write methods (POST, PUT, and DELETE) the resource might support. To show those write methods, read/write mode must be enabled.

### Read/write access mode

In the read/write access mode:

- The ArubaOS-CX REST API Reference shows all supported read and write methods for all switch resources.
- The REST API can access and change every configurable aspect of the switch as modeled in the configuration and state database.

**CAUTION:** The REST API is powerful but must be used with extreme caution: No semantic validation is performed on the data you write to the database, and configuration errors can destabilize the switch.

# Write methods (POST, PUT) supported in read-only mode

The following switch resources support write methods (POST, PUT, or both) even when the REST API access mode is set to read-only:

- Configuration management: `*/rest/v1/fullconfigs*`
- Firmware: `*/rest/v1/firmware*`
- User login and logout:
  - `*/rest/v1/login`
  - `*/rest/v1/logout`
- Aruba Network Analytics Engine and scripts: `*/rest/v1/system/nae_scripts*`

# REST requests and accounting (audit) logs

### Requests that are logged

All REST requests—including GET requests—are logged to the accounting (audit) log.

### Accounting (audit) log URI

The URI for the accounting logs is the following:

`/rest/v1/logs/audit`

### Access authorization for accounting (audit) logs

The accounting logs can be accessed by administrators or auditors.

### Sample accounting log message for a REST request

The following is an example of an accounting log message generated when a user executed a GET request:

```
type=USYS_CONFIG msg=audit(1535741482.045:70): pid=1675 uid=0 auid=4294967295
ses=4294967295 msg='rec=ACCT_CMD op=stop timezone=UTC user=admin priv-lvl=15 auth-
method=LOCAL auth-type=LOCAL service=https-server data="http-method=GET http-uri=/
rest/v1/system/bridge/vlans/1/macs" exe="/usr/bin/hpe-restd" hostname=8320
addr=127.0.0.1 terminal=? res=success'
```

The message starts with the record type, which is specific to ArubaOS-CX. Values are the following:

**`USER_START`**

Record of a user login action.

**`USER_STOP`**

Record of a user logout action.

**`USYS_CONFIG`**

Record of a command executed by the user.

The three types of accounting log information are identified by the `msg=` element starting with the `rec=` item as follows:

- Exec is identified with: `msg='rec=ACCT_EXEC`
- Command is identified with: `msg='rec=ACCT_CMD`
- System is identified with: `msg='rec=ACCT_SYSTEM`

The user group is indicated by `priv-lvl`, which is also specific to ArubaOS-CX. Values are the following:

| Privilege level | User group |
|---|---|
| 1 | `operators` |
| 15 | `administrators` |
| 19 | `auditors` |

The value of `service` indicates which user interface was used:

**`service=shell`**

Indicates that the log entry is a result of a CLI command.

**`service=https-server`**

Indicates that the log entry is a result of a REST API request or a Web UI action.

The string value of `data` identifies the CLI command or REST API request that was executed.

For REST requests, the method and URI of the REST request is contained in the string value of `data`. For example:

```
data="http-method=GET http-uri=/rest/v1/system/bridge/vlans/1/macs"
```

# REST API version

Resources, attributes, and behaviors might differ between different versions of the REST API.

The REST API version is included in the URI used in REST requests.

In the following example, the REST API version is `v1`:

```
https://192.0.2.5/rest/v1/system
```

When URIs are included in scripts, the prefix information might be declared somewhere other than in every URI used in the script.

For example, the following code specifies REST API version v1 and the management module of an Aruba 8400 switch. This code could be changed to support an Aruba 8320 switch by changing the management module resource ID to `1%2F1`. By storing the version and other commonly used information in a variable, you can update the script by changing the value of the variable instead of executing a search and replace operation throughout the script.

```
uri1 = '/rest/v1/system/subsystems/management_module/1%2F5?' \
       'attributes=resource_utilization'
```

The version of the REST API running on the switch and the version declared in the REST request must match.

At the bottom of its window, the ArubaOS-CX REST API Reference displays the REST API version running on the switch.

For example:

# Parts of a URI

The Universal Resource Identifier (URI) is the location of a specific web resource. The two main parts of a URI are the path and the (optional) query component.

**More information**

GET method parameters on page 38

## URI path, including path parameters

The path is the part of the URI starting with the server URL and ending with the resource ID. In URIs that have a query component, the path is everything before the question mark (`?`) character. The path is a hierarchy. The forward slash (`/`) character indicates the hierarchical relationship between resources.

Because the forward slash character has special meaning, forward slash characters that are part of the URL path must be percent-encoded, with the code `%2F` representing the forward slash. For example, the following URI represents the administrative state of port 1/1/3:

```
https://192.0.2.5/rest/v1/system/ports/1%2F1%2F3?attributes=admin
```

### URI prefix

The URI prefix is the system URL and REST API version information. This information is specific to a particular switch and REST API version and is the same for every REST API request to that switch.

Script writers often create a variable for the URI prefix. Using a variable enables the writer to update a script or use the same script logic for a different switch by updating the value of the URI prefix variable.

The URI prefix contains the following:

**Server URL**

The web server address of the switch.

Examples:

- `https://192.0.2.5`
- `https://10.17.0.1`
- `https://mycompany.myswitch.com`

If Virtual Switching Extension (VSX) is enabled, you can access most resources of the peer switch from this switch by inserting `/vsx-peer` in the URI path between the server URL and the REST API and version identifier.

For example:

```
GET https://192.0.2.5/vsx-peer/rest/v1/system/vsx?attributes?oper_status
```

**REST API identifier and version**

For example: `/rest/v1`

### Path parameters

A path parameter is a part of the URI path that can vary. Typically path parameters indicate a specific instance of a resource in a collection, such as a specific VLAN in the `vlans` collection. The path can contain several path parameters. Path parameters are indicated by braces `{}`.

For example, the ArubaOS-CX REST API Reference displays the resource for specific VLAN as the following:

`/system/bridge/vlans/{id}`

When you send a request for VLAN 10, the URI you provide must substitute the VLAN ID, `10`, for the `{id}` query parameter. For example:

`/system/bridge/vlans/10`

In the ArubaOS-CX REST API Reference, you enter the value of the path parameter in the **Value** field of the **id** parameter.

**More information**

VSX peer switches and REST API access on page 35

## Query component

In many cases, the unique identification of a resource requires a URI that contains both a path and a query component. The query component is sometimes called the query string.

For example, CPU utilization is a resource represented by the following URI:

`https://192.0.2.5/rest/v1/system/subsystems/management_module/1%2F5?attributes=resource_utilization`

In a URI, the question mark (`?`) character indicates the beginning of the query component. The query component contains nonhierarchical data, and the format of the query string depends on the implementation of the RESTful API.

The query component often contains "key=value" pairs separated by the ampersand (`&`) character. Multiple attribute values are supported and are separated by commas. For example:

`https://192.0.2.5/rest/v1/system/bridge/vlans?depth=1&attributes=id,name,type`

**"Dot" notation for Network Analytics Engine URIs only**

When a URI defines a monitor in an Aruba Network Analytics Engine script, attribute values in the query string support an additional dot notation that the Network Analytics Engine uses to access additional information. For example:

`https://192.0.2.5/rest/v1/system/subsystems/management_module/1%2F5?attributes=resource_utilization.cpu`

This dot notation is supported for certain URIs that define monitors in Network Analytics Engine scripts only.

**More information**

Categories of resource attributes on page 15

# Resources

In a RESTful API, the primary representation of data is called a **resource**. Resources are nouns—anything that can be named can be a resource. In a RESTful API, a resource is a representation of an entity in the system as a URI. The URI might or might not include a query component. The entities can include hardware objects, statistical information, configuration information, and status information.

Examples of resources:

- The resource utilization information

  `https://192.0.0.5/rest/v1/system/subsystems?attributes=resource_utilization`

- A list of configured VLANs:

  `https://192.0.2.5/rest/v1/system/bridge/vlans`

- The administrative state of port 1/1/3:

  `https://192.0.2.5/rest/v1/system/ports/1%2F1%2F3?attributes=admin`

- The list of all users:

  `https://192.0.2.5/rest/v1/system/users`

- The user with the ID: `myadmin`:

  `https://192.0.2.5/rest/v1/system/users/myadmin`

- The secondary firmware image:

  `https://192.0.2.5/rest/v1/firmware?image=secondary`

## Resource collections and singletons

### Collections

A collection is a directory of resources managed by the server. Typically, a resource collection contains multiple resource instances and the collection name is in the plural form.

For example:

- `/system/bridge/vlans`
- `/system/users`
- `/fullconfigs`

A GET request to a collection returns a list of the URIs of the members of the collection. The following example shows the GET request and returned response for the `vlans` collection:

```
$ curl GET -k -b /tmp/auth_cookie "https://192.0.2.5/rest/v1/system/bridge/vlans"
[
  "/rest/v1/system/bridge/vlans/1",
  "/rest/v1/system/bridge/vlans/10",
  "/rest/v1/system/bridge/vlans/20"
]
```

The brackets (`[` and `]`) enclose the list. Each URI in the list represents a configured VLAN.

To get the JSON data for VLAN 10, you must either send the GET request to the URI representing VLAN 10 (`"/rest/v1/system/bridge/vlans/10"`), or you must use the depth parameter to expand the list of URIs in the `vlans` collection to get the JSON data for all the VLANs in the collection.

### Subcollections

A single resource instance can also contain subcollections of resources.

- In the following example, `vlans` is a subcollection of the `bridge` resource:

  `/system/bridge/vlans`

- In the following example, `routes` is a subcollection of the `default` VRF resource instance:

  `/system/vrfs/default/routes`

### Singletons

There are some resources that can only have one instance. These resources are called singletons and the resource collection name is in the singular form.

For example:

- `/system`
- `/system/bridge`
- `/system/mclag`
- `/firmware`

Because there is only one resource in a singleton collection, GET requests return the JSON representation of the resource instead of a URI list of one item. In addition, you do not need to supply a resource ID in the URL of a GET request. For example, a GET request to the firmware URI returns the JSON data that represents the firmware resource:

```
$ curl GET -k -b /tmp/auth_cookie "https://192.0.2.5/rest/v1/firmware"
{
  "current_version": "TL.10.00.0006E-686-g4a43ab9",
  "primary_version": "TL.10.00.0006E-686-g4a43ab9",
  "secondary_version": "",
  "default_image": "primary",
  "booted_image": "primary"
}
```

### More information

Depth parameter on page 39

# Categories of resource attributes

Resources can contain many attributes, so those attributes are organized into categories to enable more efficient management:

**Configuration attributes**

Configuration attributes represent user-owned data. This data can be written by users through REST requests or through the switch CLI.

**Status attributes**

Contains system-owned data such as the admin account and various status fields. Users cannot create or modify instances of attributes in this category.

**Statistics attributes**

Contains system-owned data such as counters. Users cannot create or modify instances of attributes in this category.

### Attribute categories might vary

A given attribute is not necessarily in the same category from resource to resource or even resource instance to resource instance. If the system owns an instance of a resource, attributes of that resource—that might be configuration attributes if a user owns the resource instance—are status attributes that cannot be modified by users.

For example, a user can create VLANs, however, the system can also create VLANs. System-owned VLANs have many attributes that are considered to be in the status category and not the configuration category. The status category is used when the data is owned by the system and cannot be overwritten by a user.

Often a resource has a single attribute that indicates whether the resource is owned by the system or by a user. For example, for a VLAN, the `type` attribute indicates whether the VLAN was created by a user.

When this indicator attribute indicates that the resource is owned by the system, the other attributes that might have been in the configuration category are categorized as status attributes. Likewise, when the indicator attribute indicates that the resource is owned by a user, the other configuration attributes remain available for modification by users. In other words, the categories for other attributes on the resource follow the indicator attribute.

**Not all configuration attributes can be modified**

Although an attribute must be in the configuration category to be modified by a user, not all attributes in the configuration category can be modified after the resource instance is created. Configuration attributes that cannot be changed after the resource is created are called **immutable** attributes. This distinction matters when using a PUT request, because immutable attributes cannot be included in the request body.

For example, a VLAN ID is an immutable attribute. You cannot change the ID of the VLAN after the VLAN is created. The VLAN name, in contrast, is a mutable attribute. You can change the VLAN name after the VLAN is created.

**More information**

PUT method usage and considerations on page 45

# Why to use a REST API

Python and the REST API to the ArubaOS-CX database provide powerful tools to support network automation. For example, Aruba Network Analytics Engine (NAE) scripts are written in Python and use the REST API and resource URIs to identify and monitor switch resources. The NAE provides a built-in Python interpreter.

Python is the go-to language for network engineers:

- Python is high-level and human-readable.
- Python is popular with an active development community.
- There are many libraries (code written for you that you can use in your programs) available.

By using Python and the REST API, you can move far beyond CLI scripting in network automation. As an application programming interface, the REST API is optimized for machine-to-machine interactions. In contrast, a CLI is optimized for human-to-machine interactions.

In the past, a network engineer might have used Perl scripts to automate show and configure CLI commands. This scheme provided some automation, but it was inefficient because it still used the CLI to interact with the switch. CLI inputs and outputs are in an unstructured, human-readable format. You must use text processing based on specific CLI output to extract the information you want.

For example, you would have to write code to detect a MAC address in the large continuous string of text that is the CLI output. The CLI output might have many things to make it human-readable, such as table formatting, column headings, and introductory text. The way MAC addresses are presented can vary by CLI, operating system version, and sometimes even by individual command output. In addition, minor changes in CLI output from software release to software release might require you to change multiple existing scripts.

In contrast, by using programmatic interfaces such as the ArubaOS-CX REST API:

- You can get the information you are looking for in a predictable and structured way. For example, you can ask for and get just the MAC address.
- Error conditions are presented in a predictable and structured way. For example, if you execute an API to set the hostname to a new value, the results of the API call provides your program or script with a verification that it successfully executed the request. The success or failure of the operation is signaled to your program

through response messages and status codes from the API call. Your program can use these result codes to trigger the execution of other business logic.

- Changes to APIs from software release to software tend to be specific, controlled, and identified using API version information. In contrast, changes in formatting and layout of CLI output can be more frequent and difficult to detect.

For more information about Python, see: **www.python.org**

For more information about the Aruba Network Analytics Engine (NAE) and obtaining examples of NAE scripts written in Python, see the *ArubaOS-CX Network Analytics Engine Guide* for your software release.

# ArubaOS-CX REST API reference summary

The following information is intended as a quick reference for experienced users. Values are not configurable unless noted otherwise.

### Switch REST API access default

Disabled

### Switch REST API read/write access default

Read-only

### Enabling access to the Web UI and REST API

To enable access on the specified VRF, use the following CLI command from the global configuration context: `https-server vrf <VRF-NAME>`

For example:

```
switch(config)# https-server vrf mgmt
```

### Setting the REST API access mode to read/write

Use the following CLI command from the global configuration context: `https-server rest access-mode read-write`

For example:

```
switch(config)# https-server rest access-mode read-write
```

### Showing the REST API access configuration

Use the following CLI command: `show https-server`

For example:

```
switch(config)# show https-server

HTTPS Server Configuration
--------------------------

 VRF               : default, mgmt
 REST Access Mode  : read-only
```

### ArubaOS-CX REST API Reference URL:

`https://<IP-ADDR>/api/`

*<IP-ADDR>* is the IP address or hostname of your switch.

For example: `https://192.0.2.5/api/`

### REST API version

| Switch software version | REST API version |
|---|---|
| ArubaOS-CX 10.00 through 10.02 | v1 |

### Protocol

HTTPS

### Port

443

### Request and response body format

JSON

### Session idle timeout

20 minutes

### Session hard timeout

Eight hours, regardless of whether the session is active.

### Authentication

Session cookie from successful HTTPS login request.

### HTTPS client sessions

- Maximum of 48 sessions per switch.
- Maximum of three concurrent client sessions per user.
- The same session cookie is shared across browser tabs and, depending on the browser, multiple browser windows.
- The same session cookie is not shared across devices and scripts.

  For example, if a user logs into the Web UI from a laptop, again with a tablet, and then uses the same user name in a curl command, that user has three concurrent client sessions.

### VSX peer switch access

If Virtual Switching Extension (VSX) is enabled on both switches and the ISL is up, you can access the VSX peer switch from your connected switch. To access the peer VSX switch, insert `/vsx-peer` in the URI path after the server URL and before the REST API and version identifier. Not supported for login, Web UI, or ArubaOS-CX REST Reference access.

For example:

- Accessing a VSX switch:

  `https://192.0.2.5/rest/v1/…`
- Accessing its VSX peer switch:

  `https://192.0.2.5/vsx-peer/rest/v1/…`

The ArubaOS-CX Web UI and ArubaOS-CX real-time notifications subsystem rely on the REST API, so all three are enabled or disabled together.

To access the REST API, Web UI, or notifications subsystem, HTTPS server must be enabled on the specified VRF. The VRF you specify determines from which network the features can be accessed. You can enable access on multiple VRFs, including user-defined VRFs.

**Prerequisites**

- You must be in the global configuration context: `switch(config)#`
- The password for the `admin` user must be configured on the switch.

**Procedure**

1. Enable HTTPS server access for the specified VRF.

   For example:

   - To enable access on all data ports on the switch, specify the default VRF:

     ```
     switch(config)# https-server vrf default
     ```

   - To enable access on the OOBM port (management interface IP address), specify the management VRF:

     ```
     switch(config)# https-server vrf mgmt
     ```

   - To enable access on ports that are members of the VRF named `vrfprogs`, specify `vrfprogs`:

     ```
     switch(config)# https-server vrf vrfprogs
     ```

   - To enable access on the management port and ports that are members of the VRF named `vrfprogs`, enter two commands:

     ```
     switch(config)# https-server vrf mgmt
     switch(config)# https-server vrf vrfprogs
     ```

If the switch responds with the following error, the `admin` user must set a valid password:

```
Failed to enable https-server on VRF mgmt. 'admin' password is not set
```

The switch is shipped from the factory with a default user named `admin` without a password. The `admin` user must set a valid password before HTTPS servers can be enabled.

**More information**

https-server vrf on page 83

# Setting the REST API access mode to read/write

Enabling the read/write mode on the REST API allows write operations (POST, PUT, and DELETE) to be called on all configurable elements in the switch database.

The REST API in read/write mode is intended for use by advanced users who have a good understanding of the system schema and data relationships in the switch database.

**CAUTION:** Because the REST API in read/write mode can access every configurable element in the database, it is powerful but must be used with extreme caution: No semantic validation is performed on the data you write to the database, and configuration errors can destabilize the switch.

Setting the access mode is independent from enabling or disabling access to the REST API.

### Prerequisites

You must be in the global configuration context: `switch(config)#`

### Procedure

Set the REST API access mode to `read-write`.

```
switch(config)# https-server rest access-mode read-write
```

### More information

Considerations when making configuration changes on page 43
Enabling access to the REST API on page 19
https-server rest access-mode on page 82

# Showing the REST API access configuration

### Procedure

To show the REST API access configuration, in the manager context (`#`) of the CLI, enter the `show https-server` command.

For example:

```
switch# show https-server
    HTTPS Server Configuration
    --------------------------
     VRF               : mgmt, default
     REST Access Mode  : read-write
```

The command output lists the VRFs on which access to REST API is enabled and shows the current REST API access mode.

If access is not enabled on any VRF, the VRF configuration is displayed as `<none>`.

For example:

```
switch# show https-server
    HTTPS Server Configuration
    --------------------------
     VRF               : <none>
     REST Access Mode  : read-write
```

### More information

show https-server on page 84

# Disabling access to the REST API

**NOTE:** The ArubaOS-CX Web UI and ArubaOS-CX real-time notifications subsystem rely on the REST API, so all three are enabled or disabled together.

### Prerequisites

You must be in the global configuration context: `switch(config)#`

### Procedure

Disable HTTPS server access for the specified VRF by using the `no` form of the `https-server vrf` command.

For example, the following command disables REST API access on the switch data ports in the default VRF:

```
switch(config)# no https-server vrf default
```

You can use the `show https-server` command to show the current configuration:

```
switch# show https-server

HTTPS Server Configuration
--------------------------

 VRF               : mgmt
 REST Access Mode  : read-write
```

You can access the REST API using any REST client interface that supports HTTPS requests and supports obtaining and passing a session cookie.

Examples of client interfaces include the following:

**Scripts and programs that support HTTPS requests**

The most powerful way to access the ArubaOS-CX REST API is to use a programming language that supports HTTPS requests, such as Python, to write programs that automate network management tasks.

**The curl command-line interface**

You can use curl commands either interactively or within a script to make REST requests. Using curl commands can be a way to execute GET requests without writing a script. Using curl commands can be a way to test REST requests that you are considering incorporating into an application.

**Browser-based interfaces such as Postman or the ArubaOS-CX REST API Reference**

Examples of browser-based interfaces include Postman and the ArubaOS-CX REST API Reference.

The ArubaOS-CX REST API Reference documents the switch resources, parameters, and JSON models for each HTTPS method supported by the resource. Because the ArubaOS-CX REST API Reference is browser-based, it can share the session cookie with a Web UI session active in another browser tab. The ArubaOS-CX REST API Reference is not intended to be used as a configuration tool and is not required for day-to-day operations.

The ArubaOS-CX REST API Reference is one way to execute GET requests without writing a script. The ArubaOS-CX REST API Reference can be used during script coding to help you construct the URIs—with their query parameters—that you use in a script or curl command.

# Accessing the REST API using the ArubaOS-CX REST API Reference

**CAUTION:** Although the ArubaOS-CX REST API Reference interacts directly with the REST API, the ArubaOS-CX REST API Reference is not intended as a management or configuration interface. Use caution when using the **Submit** button for POST or PUT methods because this action can result in changes to your current environment.

**Prerequisites**

- HTTPS server access must be enabled.
- With a few exceptions, using the PUT, POST, or DELETE methods require the following conditions to be true:
  - The REST API access mode must be set to `read-write`.
  - The user name you use to log in must be a member of the `administrators` group.

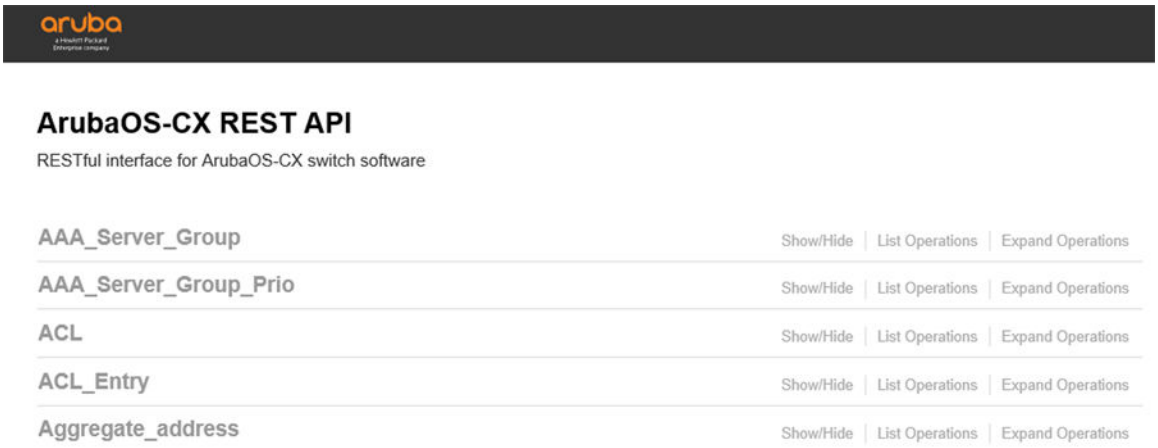
**Procedure**

Open a browser at: `https://<IP-ADDR>/api/`

*<IP-ADDR>* is the IP address or hostname of your switch.

For example: `https://192.0.2.5/api/`

The web browser displays a page similar to the following example (partial page shown):




## Logging in and logging out using the ArubaOS-CX REST API Reference

**Prerequisites**

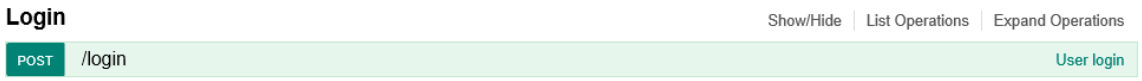
- Access to the switch REST API must be enabled.
- You must have used a supported browser to access the switch at:

  `https://<IP-ADDR>/api/`

  *<IP-ADDR>* is the IP address or hostname of your switch.

**Procedure**

1. Log in to the switch using the **Login** resource:

   a. Expand the **Login** resource by clicking the resource name, `/login`, or by clicking **Expand Operations**.

   b. Enter your user name in the value of the **username** parameter.

   c. Enter your password in the value of the **password** parameter.

   d. Click **Submit**.

   If the operation is successful, the REST API returns response code 200.

2. When you finish your session, log out by expanding the **Logout** resource and clicking **Submit**.

# ArubaOS-CX REST API Reference basics

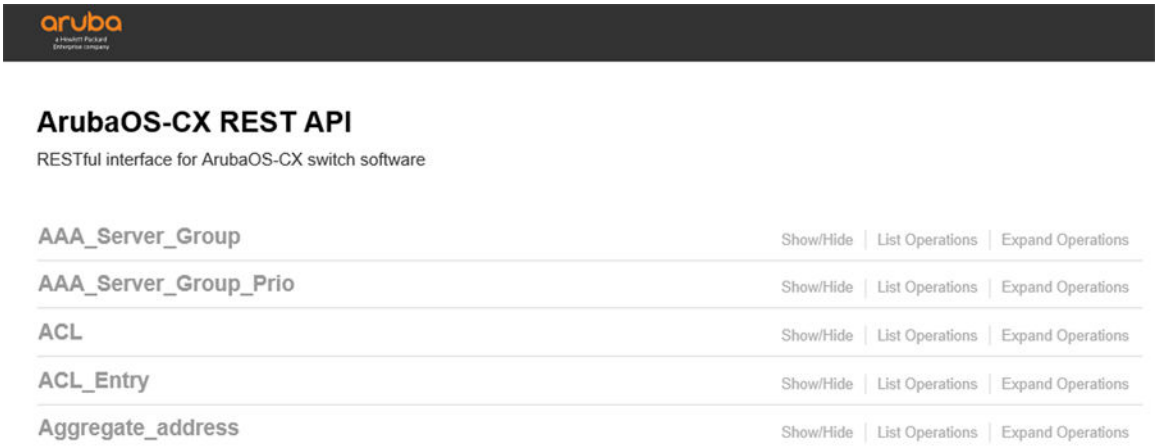
The ArubaOS-CX REST API Reference is a web interface based on Swagger 2.0. The ArubaOS-CX REST API Reference does the following:

- Documents the switch REST API resources, methods, models, and responses.
- Enables you to interact with the switch to view or change the configuration.

For more information about Swagger, see **https://swagger.io/**.

### ArubaOS-CX REST API Reference home page

The following is an example of a portion of the ArubaOS-CX REST API Reference home page for a switch running ArubaOS-CX software:



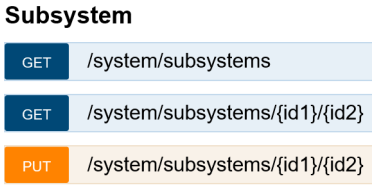

At the bottom of the page, the ArubaOS-CX REST API Reference shows base URL and version information. For example:

```
[ BASE URL: /rest/v1, API VERSION: 1.0.0 ]
```

- The switch resource URIs are organized in groups. The group names are listed in alphabetical order on the ArubaOS-CX REST API Reference home page.

  The group name does not always match the resource collection name. Use the group names as a navigation aid only.
- Group names that are in gray have the URI entries—also called endpoints—collapsed. When you hover over the group name, it turns black. Click the group name to expand it and show the list of methods and URIs in the group.

  You can also use the **Show/Hide**, **List Operations**, and **Expand Operations** controls to expand or collapse all the members of the group.
- The following example shows the list of the methods and resource URIs in the Subsystem group:

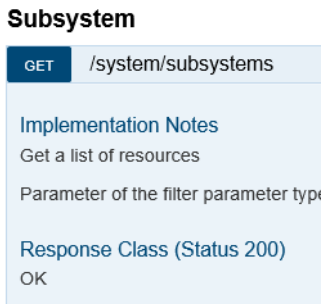
  
  

  This view is the same view that is shown when you click the **List Operations** control of the **Subsystem** group.
  - The methods that are shown might depend on the REST API access mode. Some methods might not be displayed if the REST API access mode is `read-only`.
  - Methods and resources might be displayed that you do not have the authorization to access. For example, users with operator rights are not authorized to make PUT or POST requests to most resources. If you submit a request for which you are not authorized, the switch returns the following error:

```
HTTP error 403 "Forbidden"
```

- The resource collection name is `subsystems` (not Subsystem).
- Items in braces, such as `{id 1}`, are path parameters. If you submit a request to a resource URI that includes a path parameter, you are required to supply a value for the parameter.

- To show more information about an item on the list, click the URI path. The following shows part of the information displayed when /system/subsystems is selected:




You can use the browser scroll bar to navigate to information about the implementation of this method and resource, including the required and optional parameters.

- Required parameters are shown in **bold**.

  For example, the POST method of the login resource requires a user name and password:

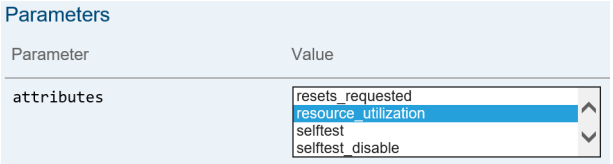

  Path parameters such as `{id}` are listed as required parameters:


- The **Submit** button sends the request.

**CAUTION:** Although the ArubaOS-CX REST API Reference interacts directly with the REST API, the ArubaOS-CX REST API Reference is not intended as a management or configuration interface. Use caution when using the **Submit** button for POST or PUT methods because this action can result in changes to your current environment.

- In GET requests, there can be multiple attributes and parameters you can use to filter results.

  For example:




  You can select multiple attributes:

- To select a range of attributes, click the first attribute, then press **Shift**, and then click the last attribute in the range you want to select.
- To select attributes that are not adjacent in the list, press **Ctrl**, then click each attribute you want to select.

- The JSON model for the resource is described in **Model** and shown with example values in **Example Values** for each method. The following example shows the JSON model and example values for PUT method of the `/system/subsystems/{id1}/{id2}` resource:


- After a request is submitted, the ArubaOS-CX REST API Reference shows additional information, including the following:
  - The curl command equivalent of the submitted request
  - The submitted request URL, including the specified parameters and values.
  - The response body returned by the switch
  - The response code returned by the switch
  - The response headers returned by the switch
- The curl command and request URLs are displayed using percent encoding for certain characters in the query string portion of the URL:

| Character | Percent-encoded equivalent |
|---|---|
| , (comma) | %2C |
| : (colon) | %3A |

When you enter curl commands or submit requests through other means, percent encoding is permitted but not required in the query string of the URI.

# Accessing the REST API using curl

When you use curl, you log in at the beginning of your session and log out at the end of the session. When you log in, you must save the cookie returned from the login request so that you can pass that same cookie value to the switch in subsequent curl commands.

### Prerequisites

Access to the switch REST API must be enabled.

### Procedure

1. To access the ArubaOS-CX REST API using curl, use curl version 7.35 or later.
2. For all curl commands, use the `-k` option to disable certificate validation.

   The switch uses self-signed certificates. By default, the curl program attempts to verify certificates against its list of certificate authorities, and attempts to verify self-signed certificates fail. Therefore you must use the `-k` option to disable attempts to verify self-signed certificates against a certificate authority.
3. Start your session by logging in. When you log in, save the cookie file by specifying the `-c` option with a file name.
4. In all subsequent curl commands—including logging out—pass the cookie value back to the switch by specifying the `-b` option with the same file name.
5. At the end of the session, log out of the switch using curl.

**IMPORTANT:** Logging out at the end of the session is important because the number of concurrent HTTPS sessions per client and per switch are limited, and session cookies are not shared across devices and scripts.

## Logging in using curl

### Prerequisites

Access to the switch REST API must be enabled.

**CAUTION:** Credential information (user name, password, domain, and authentication tokens) used in curl commands entered at a command-line prompt might be saved in the command history. For security reasons, Hewlett Packard Enterprise recommends that you disable command history before executing commands containing credential information.

### Procedure

Use the following curl command to access the `login` resource of the switch and provide your user name and password as data:

Syntax:

```
curl -k[ --noproxy <IP-ADDR>] POST
-c <COOKIE-FILE>
-H 'Content-Type: application/x-www-form-urlencoded'
"https://<IP-ADDR>/rest/v1/login"
--data 'username=<USER-NAME>&password=<PASSWORD>'
```

**`-k`**

Specifies that the curl program not attempt to verify the server certificate against the list of certificate authorities included with the curl software.

The switch uses self-signed certificates. By default, the curl program attempts to verify certificates against its list of certificate authorities, and attempts to verify self-signed certificates fail. Therefore you must use the `-k` option to disable attempts to verify self-signed certificates against a certificate authority.

**`--noproxy <IP-ADDR>`**

Optional. The `--noproxy` option is appropriate where execution of curl commands does not need a proxy to access the applications. If your network is configured to require a proxy to access applications, use the `--proxy` option. *<IP-ADDR>* specifies the IP address or hostname of the switch.

**`-c <COOKIE-FILE>`**

Specifies the file in which to store the session cookie. This session cookie is required when you execute subsequent curl commands.

***<USER-NAME>***

Specifies the user name.

***<PASSWORD>***

Specifies the password for the user.

**NOTE:** Although it is possible to pass the user name and password information as a query string in the login URI, system logs save the accessed URI in cleartext in log entries. Hewlett Packard Enterprise recommends that you pass the credential information as data instead of in the URI when using programs such as curl to log in to the switch.

Example:

```
$ curl --noproxy 192.0.2.5 -k POST \
-c /tmp/auth_cookie \
-H 'Content-Type: application/x-www-form-urlencoded' \
"https://192.0.2.5/rest/v1/login" \
--data 'username=admin&password=admin'
```

## Logging out using curl

**Procedure**

Use the following curl command to access the `logout` resource of the switch:

Syntax:

```
curl -k[ --noproxy <IP-ADDR>] -X POST
-b <COOKIE-FILE>
"https://<IP-ADDR>/rest/v1/logout"
```

**`-k`**

Specifies that the curl program not attempt to verify the server certificate against the list of certificate authorities included with the curl software.

The switch uses self-signed certificates. By default, the curl program attempts to verify certificates against its list of certificate authorities, and attempts to verify self-signed certificates fail. Therefore you must use the `-k` option to disable attempts to verify self-signed certificates against a certificate authority.

**`--noproxy <IP-ADDR>`**

Optional. The `--noproxy` option is appropriate where execution of curl commands does not need a proxy to access the applications. If your network is configured to require a proxy to access applications, use the `--proxy` option. *`<IP-ADDR>`* specifies the IP address or hostname of the switch.

**`-b <COOKIE-FILE>`**

Specifies the file that contains the session cookie.

**NOTE:** When you use curl, you log in at the beginning of your session and log out at the end of the session. When you log in, you must save the cookie returned from the login request so that you can pass that same cookie value to the switch in subsequent curl commands. When you log in, save the cookie file by specifying the `-c` option with a file name.

In subsequent curl commands, pass the cookie value back to the switch by specifying the `-b` option with the same file name.

Example:

```
$ curl -k --noproxy 192.0.2.5 -X POST \
-b /tmp/auth_cookie \
"https://192.0.2.5/rest/v1/logout"
```

## The curl command-line application

There are several tools available for accessing RESTful web service APIs, one of which is curl. The curl tool, created by the cURL project, is a command-line application for transferring data using URL syntax.

For details on installing the curl application, see **https://curl.haxx.se/download.html**.

The curl application has many options, which are described in detail in the curl manual (run "curl --manual") and at **https://curl.haxx.se/docs/manpage.html**.

## About the curl command examples

In the curl examples, the workstation is running a Linux-based operating system and curl version 7.35 is installed.

The curl examples generated by the ArubaOS-CX REST API Reference might use different options than in other examples, and do not include cookie file handling because the cookie is handled by the browser.

Many examples of curl commands are formatted in multiple lines for readability. The backslash (`\`) continuation character at the end of the line indicates that the command continues on the next line.

The curl command examples in this document use minimal options. The following options are commonly used in the curl command examples:

**`-b <cookie-file>`**

Specifies that the file *`<cookie-file>`*, which contains the session cookie, be passed with the request. *`<cookie-file>`* specifies the path and name of the cookie file.

When you use curl, you log in at the beginning of your session and log out at the end of the session. When you log in, you must save the cookie returned from the login request. You must provide the cookie with every subsequent curl command.

**`-k`**

Specifies that the curl program not attempt to verify the server certificate against the list of certificate authorities included with the curl software.

The switch uses self-signed certificates. By default, the curl program attempts to verify certificates against its list of certificate authorities, and attempts to verify self-signed certificates fail. Therefore you must use the `-k` option to disable attempts to verify self-signed certificates against a certificate authority.

**`--noproxy`**

Specifies that a web proxy is not required. The `--noproxy` option is appropriate where execution of curl commands does not need a proxy to access the applications.

If your network is configured to require a proxy to access applications, use the `--proxy` option instead of the `--noproxy` option.

**`-d '<string>'`**

Specifies that curl send the data in *<string>* in a POST request using the content-type application/x-www-form-urlencoded.

**`-X`**

Specifies a method that curl would not use by default. Typically used with PUT or DELETE methods only.

**`-H` or `--header <header>`**

Specifies an extra header in the HTTP request.

**`-D-`**

Specifies that curl write the returned protocol headers to the standard output file. Used for debugging.

More options can be used to customize your experience for your environment. For more information about curl options, see:

**https://curl.haxx.se/docs/manpage.html**

# Accessing the REST API using a Python script

**Procedure**

- To send requests to the REST API using a Python script, Hewlett Packard Enterprise recommends that you use an HTTP library.

  For example, the `requests` library is an HTTP library for Python that you can install using the following PIP command:

  ```
  pip install requests
  ```

  PIP is included with 3.4 or later version of Python.

  After you have installed the library, you can import the library into your Python script.

  Usage example:

  ```
  import requests
  ...
  r = requests.get('https://api.github.com/events')
  r.json()
  ...
  ```

  For more information, see:

  **http://docs.python-requests.org/en/master/**

- In the part of the code that handles logging in, disable warnings about insecure requests.

  For example:

  ```
  # disable warnings that Web UI site is "insecure"
  # needed to avoid connection timeout due to inability to verify server certificate
  ```

```
from requests.packages.urllib3 import disable_warnings
from requests.packages.urllib3.exceptions import InsecureRequestWarning
disable_warnings(InsecureRequestWarning)
```

If you do not disable warnings about insecure requests, the request to log in hangs until the request times out because the server certificate could not be verified. An error similar to the following is returned:

```
Connection timed out
```

- In all requests, use the `cookies` parameter to pass the session cookie to the switch, and the `verify=False` parameter to disable server certificate verification.

  For example

  ```
  response = requests.post(
          url=url,
          cookies=self.cookie,
          verify=False,
      )
  ```

  To avoid connection timeout errors because of the use of self-signed certificates, the Python script must include **both** the `disable_warnings(InsecureRequestWarning)` and `verify=false` in all requests.

  The session cookie is required to authenticate the request.

- Ensure that the script logs out of the switch before exiting.

  **IMPORTANT:** Logging out at the end of the session is important because the number of concurrent HTTPS sessions per client and per switch are limited, and session cookies are not shared across devices and scripts.

- You can program the process of logging in, storing and passing the session cookie, and error handling in a variety of ways.

  For a detailed example that handles errors and response codes, see **Example: Logging in and logging out using Python** on page 31.

# Example: Logging in and logging out using Python

The following example shows the parts of a Python program that handle the following:

- Logging in
- Logging out
- Response codes
- Disabling server certificate verification
- Storing and passing the session cookie

```
# library to take inline args
import sys

# library to make REST requests
import requests

# regular expressions library
import re

# disable warnings that Web UI site is "insecure"
```

```
# needed to avoid connection timeout due to inability to verify server certificate
from requests.packages.urllib3 import disable_warnings
from requests.packages.urllib3.exceptions import InsecureRequestWarning
disable_warnings(InsecureRequestWarning)

if len(sys.argv) is not 4:
    print("usage: 8400rest.py <switch_ip> <username> <password>")
    print("note: admin access is required to edit data")
    exit(0)

# parameters
ver = 'v1'
sw_ip = sys.argv[1]
usr = sys.argv[2]
pswd = sys.argv[3]

def main():
    # class instance
    sw = sw_rest(ip=sw_ip, ver=ver)

    # only perform actions if login is successful
    if (sw.login(user=usr, passwd=pswd)):
        #
        # Script action calls go here
        #
        sw.logout()


# class for rest calls to ArubaOS-CX switch
class sw_rest:

    def __init__(self, ip=None, ver=None):
        """
        init class function

        :param str ip: ip address of switch to access (required)
        :param str ver: version of ArubaOS-CX REST (default 'v1')
        """
        # check and set ip address
        if (ip is None):
            print("Switch IP address is required")
            exit(1)
        else: self.ip = ip

        # check and set REST version
        if (ver is None):
            print("REST version not specified... using v1")
            self.ver = 'v1'
        else: self.ver = ver

        # create base address for REST requests (https://<ip>/rest/<ver>/)
        self.base_url = '/'.join(['https:/', self.ip])
        self.base_uri = '/'.join(['', 'rest', self.ver, ''])
        self.base = self.base_url + self.base_uri

        # cookie jar to store cookies after login
        self.cookie = requests.cookies.RequestsCookieJar()
```

```
def login(self, user=None, passwd=None):
    """
    login to switch

    :param str user: username for login (required)
    :param str passwd: password for login (required)
    :return: whether login was successful
    :rtype: bool
    """
    self.login = {'username': user, 'password': passwd}
    # url = https://<ip>/rest/<ver>/login
    url = self.base + 'login'

    print("Logging in...")
    response = requests.post(
            url=url,
            params=self.login,
            verify=False,
        )
    print(response)

    # determine if login was successful
    status = self.status_handler(response.status_code)

    # set cookie if login is successful
    if (status):
        self.cookie.set('id', response.cookies['id'], domain=self.ip)

    return status

def logout(self):
    """
    logout of switch

    :return: whether logout was successful
    :rtype: bool
    """
    # url = https://<ip>/rest/<ver>/logout
    url = self.base + 'logout'

    print("Logging out...")
    response = requests.post(
            url=url,
            cookies=self.cookie,
            verify=False,
        )
    print(response)

    return self.status_handler(response.status_code)



def status_handler(self, status_code):
    """
    determine if the response is good or bad
```

```
        :param int status_code: status code of response
        :return: whether response is good
        :rtype: bool
        """
        return (status_code == requests.codes.ok)

if __name__ == "__main__":
    main()
```

If Virtual Switching Extension (VSX) is enabled, you can access the REST API of a peer switch without having to separately log into or manage a session cookie from that peer switch.

To access a peer REST API from your connected switch, insert `/vsx-peer` in the URI path after the server URL and before the REST API and version identifier.

For example:

`https://192.0.2.5/vsx-peer/rest/v1/...`

Notes:

- VSX must be enabled on both switches, and the interswitch link (ISL) must be up.
- REST API access must be enabled on both switches.
- For write access, the REST API access mode must be set to read/write on both switches.
- You must be logged in to the switch to which you are connected. For example, if you are connected to the primary VSX switch, you must be logged in to the primary switch.
- The following uses of `/vsx-peer` in the URI path are not supported:
  - The login resource. Requests to `/vsx-peer/rest/v1/login` are not supported.
  - Accessing the Web UI. Setting the browser address to `https://<connected_switch_ip>/vsx-peer` is not supported.
  - Accessing the real-time notifications framework through the WebSockets connection. Setting the connection to address to `wss://<connected_switch_ip>/vsx-peer/rest/v1/notification` is not supported.

    However, you can specify the peer VSX switch when you subscribe to topics.
- Audit messages are logged on the peer device.
- When configuration synchronization is enabled, supported configuration changes on the primary VSX switch are replicated on the secondary VSX switch. Changing the configuration of a secondary VSX switch might cause the configurations to be out of synchronization.

For more information about VSX, see the *ArubaOS-CX Virtual Technologies Guide* for your software release.

Examples:

- Getting the VSX status of the secondary VSX switch while connected to the primary VSX switch at IP address 192.0.2.5:

```
$ curl -k --noproxy 192.0.2.5 GET \
-b /tmp/primary_auth_cookie \
"https://192.0.2.5/vsx-peer/rest/v1/system/vsx?attributes?oper_status"
```

- Getting the VSX status of the primary VSX switch while connected to the secondary VSX switch at IP address 192.0.2.6:

```
$ curl -k --noproxy 192.0.2.6 GET \
-b /tmp/secondary_auth_cookie \
"https://192.0.2.6/vsx-peer/rest/v1/system/vsx?attributes?oper_status"
```

- Getting the name and IP addresses of ports on secondary VSX switch while connected to the primary VSX switch at IP address 192.0.2.5:

```
$ curl -k --noproxy 192.0.2.5 GET \
-b /tmp/primary_auth_cookie \
"https://192.0.2.5/vsx-peer/rest/v1/system/ports?attributes=name,ipv4_adress"
```

The GET method is a read method that gets the resource specified by the URI.

Data is returned in JSON format in the response body.

### GET on a resource collection returns a list of resource URIs

Using GET on a resource collection results in a list of URIs. Each URI in the list corresponds to a specific resource in the collection.

The following example shows a GET request to the `vlans` collection. The response body is a list—in JSON format—of the configured VLANs. Each VLAN resource is listed in URI format.

```
$ curl GET -k -b /tmp/auth_cookie "https://10.17.0.1/rest/v1/system/bridge/vlans"
[
  "/rest/v1/system/bridge/vlans/1",
  "/rest/v1/system/bridge/vlans/10",
  "/rest/v1/system/bridge/vlans/20"
]
```

### GET on a resource returns the attributes of that resource

GET on a specific resource returns the attributes of that resource.

The following example shows a GET request on VLAN 10. The URI for VLAN 10 specifies resource ID 10 in the `vlans` collection.

The response body is the representation of VLAN 10 in JSON format. In response bodies, references to other resources are represented as URIs. For example, VLAN 10 has an attribute called `flood_enabled_subsystems`, that has a value that is a list of subsystems represented as a URI.

```
$ curl GET -k -b /tmp/auth_cookie "https://10.17.0.1/rest/v1/system/bridge/vlans/10"
{
  "id": 10,
  "name": "vlan10",
  "type": "static",
  "flood_enabled_subsystems": [
    "/rest/v1/system/subsystems/system/base"
  ],
…
}
```

# Wildcard character support

When you use the GET method, the URI can contain the asterisk (`*`) wildcard character instead of a component in URI path. You can use wildcard characters in multiple places in the path. You cannot use a wildcard character as part of the query string.

The wildcard character must replace the entire component in the path. For example, you can use a wildcard to specify all VRFs, but you cannot use a wildcard character to specify all VRFs that begin with the letter `r`.

By using a wildcard character in place of a component in the path, you can specify that GET return information about multiple resources without requiring you to name each resource instance or to execute multiple GET requests.

For example:

- The following URI specifies all routes regardless of VRF:

  `"https://192.0.2.5/rest/v1/system/vrfs/*/routes"`

- The following URI specifies all ACL entries of type IPv4, regardless of the name of the ACL:

  `"https://192.0.2.5/rest/v1/system/acls/*/ipv4/cfg_aces"`

- The following URI specifies the connection state of all BGP neighbors belonging to all BGP routers in the "red" VRF:

  `"https://192.0.2.5/rest/v1/system/vrfs/red/bgp_routers/*/bgp_neighbors/*?attributes=conn_state"`

# GET method parameters

The GET method supports the following parameters in the query string of the URI:

- `attributes`
- `count`
- `depth`
- `filter`
- `selector`

A parameter is specified as a "key=value" pair. When permitted, multiple values are separated by the comma (`,`) character.

For example:

- `attributes=id,name,type`
- `count=true`
- `depth=1`
- `filter=type:static`
- `selector=configuration`

A parameter can be used alone or in combination with other parameters. The ampersand (`&`) character separates each parameter in the string.

For example:

```
GET "https://192.0.2.5/rest/v1/system/bridge/vlans?depth=1&attributes=id,name,type"
```

# Attributes parameter

The `attributes` parameter of the GET method reduces the returned data for each entry to include only the attributes specified in the comma-separated list. The attribute names in the URI must match the attribute names in the ArubaOS-CX REST API Reference.

For a list of the available attributes for a resource, see the GET method of that resource in the ArubaOS-CX REST API Reference.

Example request:

```
GET "https://192.0.2.5/rest/v1/system/bridge/vlans?depth=1&attributes=id,name,type"
```

Example response:

```
[
  {
    "id": 1,
    "name": "DEFAULT_VLAN_1",
    "type": "default"
  },
  {
    "id": 2,
    "name": "VLAN2",
    "type": "static"
  },
  {
    "id": 3,
    "name": "VLAN3",
    "type": "static"
  }
]
```

# Count parameter

The `count` parameter of the GET method returns the number of entries that match the specified URI. The count parameter can be useful when specifying resource collections or for getting statistical information.

You can specify the count parameter as either of the following:

- `count`
- `count=true`

Examples:

- Use the `count` parameter to get the total number of VLANs:

  ```
  GET "https://192.0.2.5/rest/v1/system/bridge/vlans?count=true"
  ```

- Use the `count` parameter with the `filter` parameter to get the total number of interfaces in a `down` administrative state:

  ```
  GET "https://192.0.2.5/rest/v1/system/interfaces?count&filter=admin_state:down"
  ```

# Depth parameter

The `depth` parameter of the GET method specifies to what level URIs in response bodies are to be expanded and replaced by the JSON representation of that resource:

- Default: 0
- Maximum: 3

For each level of depth, the REST API expands one level of URIs into their JSON data representations in the response body.

**NOTE:** Using the depth parameter can result in large amounts of returned data, depending on the number of items in the list and the amount of JSON data that represents each item.

For example, a GET request on the `vlans` resource returns a list of URIs. Example request:

```
GET "https://192.0.2.5/rest/v1/system/bridge/vlans"
```

Example response:

```
[
  "/rest/v1/system/bridge/vlans/1",
  "/rest/v1/system/bridge/vlans/10",
  "/rest/v1/system/bridge/vlans/20"
]
```

To specify that those URIs also be expanded and replaced with the JSON data, specify `depth=1` as a parameter in the GET request.

Example request:

```
GET "https://192.0.2.5/rest/v1/system/bridge/vlans?depth=1"
```

Example response (ellipses represent data omitted from this example):

```
[
  {
    "id": 1,
       "name": "DEFAULT_VLAN_1",
       "type": "default",
       …
    "flood_enabled_subsystems": [
      {
         URI-of-first-subsystem
      },
      …
      {
         URI-of-last-subsystem
      }
    ]
  },
     { "id": 10,
       "name": "vlan10",
       "type": "static",
       …
    "flood_enabled_subsystems": [
      {
         URI-of-first-subsystem
      },
      …
      {
         URI-of-last-subsystem
      }
    ]
  }
]
```

Each VLAN in the preceding example includes an attribute, `flood_enabled_subsystems`, which contains a list of URIs that represent the flood-enabled systems. To specify that those URIs also be expanded and replaced with the JSON data, specify `depth=2` as a parameter in the GET request.

# Filter parameter

The `filter` parameter of the GET method reduces the returned data to include only those entries that match the filter criteria. Specify the filter criteria in a comma-separated list of attribute `name:value` pairs.

Examples:

- Use the `filter` parameter to get only the static VLANS:

  ```
  GET "https://192.0.2.5/rest/v1/system/bridge/vlans?filter=type:static"
  ```

- Use the `filter` parameter to get the BGP routes that have 1.1.1.1 as a peer:

  ```
  GET "https://192.0.2.5/rest/v1/system/vrfs/default/bgp_routes?filter=peer:1.1.1.1"
  ```

# Selector parameter

The `selector` parameter of the GET method filters the returned data to include only those attributes that belong to the specified category. By using the `selector` parameter, you avoid having to list attributes individually using the `attributes` parameter.

The default is to include all categories. Use a comma (`,`) to separate multiple category values.

The selector categories are the following:

**`configuration`**

Contains user-owned information. Attributes in the configuration category can be supplied by users through REST requests or through the switch CLI. Although an attribute must be in the configuration category to be modified by a user, not all attributes in the configuration category can be modified after the resource instance is created.

**`statistics`**

Contains system-supplied data such as counters. Attributes in the `statistics` category cannot be written by users.

**`status`**

Contains system-owned data such as the admin account and various status fields. Attributes in the `status` category cannot be written by users.

For example, to get the configuration attributes of all VLANs, when you specify the URI of the GET method, do the following:

- Specify `depth=1` to direct the REST API return the JSON representations of each VLAN instead of the URI of each VLAN in the list. If you do not specify `depth=1`, the REST API returns each VLAN represented as a URI, which does not include the attributes of the individual VLANs.
- Specify the `selector` parameter with the value `configuration`.

```
GET "https://192.0.2.5/rest/v1/system/bridge/vlans?depth=1&selector=configuration"
```

Example response:


```
[
  {
    "admin": "up",
    "id": 20,
    "mgmd_enable": {},
    "mgmd_igmp_block_ports": [],
    "mgmd_igmp_fastleave_ports": [],
    "mgmd_igmp_forcedfastleave_ports": [],
    "mgmd_igmp_forward_ports": [],
    "mgmd_igmp_static_groups": [],
    "mgmd_mld_block_ports": [],
    "mgmd_mld_fastleave_ports": [],
    "mgmd_mld_forcedfastleave_ports": [],
```

```
    "mgmd_mld_forward_ports": [],
    "mgmd_mld_static_groups": [],
    "name": "VLAN20",
    "type": "static"
  },
  {
    "admin": "up",
    "id": 10,
    "mgmd_enable": {},
    "mgmd_igmp_block_ports": [],
    "mgmd_igmp_fastleave_ports": [],
    "mgmd_igmp_forcedfastleave_ports": [],
    "mgmd_igmp_forward_ports": [],
    "mgmd_igmp_static_groups": [],
    "mgmd_mld_block_ports": [],
    "mgmd_mld_fastleave_ports": [],
    "mgmd_mld_forcedfastleave_ports": [],
    "mgmd_mld_forward_ports": [],
    "mgmd_mld_static_groups": [],
    "name": "VLAN10",
    "type": "static"
  }
]
```

The supported write methods are POST, PUT, and DELETE:

- POST creates a resource.
- PUT replaces a resource.
- DELETE removes a resource.

Not all resources support all write methods. See the ArubaOS-CX REST API Reference for the methods supported by each resource. The REST API must be in read/write mode for the ArubaOS-CX REST API Reference to show all the write methods a resource supports.

# Considerations when making configuration changes

The REST API can access and change every configurable aspect of the switch as modeled in the configuration and state database. However, changing the configuration of a switch through the REST API can be different than changing the configuration through the CLI.

A single configuration change to the switch can require changes to multiple resources in the configuration and state database. Often these changes must be made in a specific order.

The CLI commands have been programmed to work "behind the scenes" to make the correct database changes and to perform data validation checks on the user input. In contrast, when you use the REST API to make a configuration change, you must become familiar with the representational models of the switch resources, the type and format of the data required, and the required order of write operations to various resources.

**CAUTION:** The REST API is powerful but must be used with extreme caution: No semantic validation is performed on the data you write to the database, and configuration errors can destabilize the switch. Hewlett Packard Enterprise recommends that you refer to the tested examples when using the REST API to make configuration changes.

**More information**

VSX peer switches and REST API access on page 35
Examples on page 70

# Considerations for ports and interfaces

The words "port" and "interface" have meanings that are different from other network operating systems. In the ArubaOS-CX operating system:

- A port is the logical representation of a port.
- An interface is the hardware representation of a port.
- When creating a VLAN or LAG interface in the CLI, you are creating ports in the REST API.

Most resources have a hierarchical relationship. You must create the parent before you can create the child. However, ports and interfaces do not have a parent that is clear from the resource hierarchy. Therefore, there are some things you do differently from other resources when creating and deleting ports and interfaces:

- Hardware interfaces are included in the database automatically. Considerations for interfaces apply to user-created interfaces such as VLAN interfaces or LAG interfaces.
- When you insert a transceiver in the physical port, the port is automatically created. Because the port already exists, executing a POST request to create a port after a transceiver is inserted does not work.
- Before you create a VLAN or LAG interface, you must create the port the interface will be referenced by.
- When you create a port, if you choose a VRF other than the default VRF, you must create that VRF before you create the port.
- When you create a port, you must explicitly designate either a VRF or the bridge that refers to the port. The POST method JSON model for a port includes the field `referenced_by` to designate this relationship.

  The following example creates a port resource for a VLAN interface:

  ```
  POST "https://192.0.2.5/rest/v1/system/ports"
  {
    "name": "vlan200",
    "admin": "up",
    "interfaces": [],
    "referenced_by": "/rest/v1/system/vrfs/default",
    "vlan_tag": "/rest/v1/system/bridge/vlans/200",
    "description": "referenced by /system/bridge/vrfs/default"
  }
  ```

- When you create a VLAN or LAG interface, you must explicitly designate the port that refers to this interface. The POST method JSON model includes the field `referenced_by` to designate this relationship.

  The following example creates a VLAN interface:

  ```
  POST "https://192.0.2.5/rest/v1/system/interfaces"
  {
    "name": "vlan200",
    "referenced_by": "/rest/v1/system/ports/vlan200",
    "description": "referenced by /system/bridge/ports/vlan200",
    "type": "internal",
    "user_config": {
      "admin": "up"
    }
  }
  ```

- Only the POST method JSON models for ports and interfaces include the `referenced_by` field. The `referenced_by` field is not included in the response body for GET requests, and is not supported by the PUT method.
- The `ports` and `interfaces` resources do not support the DELETE method. After the port or interface is removed from all the resources that reference it, the port or interface is removed from the system automatically. This automatic removal is done through the same process that automatically removes resources that have no parent resource.

  For example, to delete the VLAN interface created in the preceding example, `"/rest/v1/system/interfaces/vlan200"`, you must delete the port from the list of ports in the referring VRF.
- Hewlett Packard Enterprise recommends that you include `referenced_by` information in the `description` field when you create the port or interface. By including this information when the port or interface is created, you avoid having to query all possible referring resources (such as all VRFs) when you want to remove the port or interface.

# POST method usage and considerations

The POST method creates an instance of a resource in the collection specified by the URI:

- Not all resources support the POST method. See the ArubaOS-CX REST API Reference for the methods supported by each resource. The REST API must be in read/write mode to see all the POST methods supported.
- Some resources support the POST method even when the REST API is in read-only mode.
- When you use the POST method, the URI must point to the collection—not to the resource you are creating. The resource you are creating is sent in JSON format in the request body.
  - The JSON representation must include all fields required by the JSON model of that resource.
  - The JSON representation can contain only configuration attributes. It must not contain attributes in the `status` or the `statistics` category.
- You can POST only one resource at a time.
- Most resources have a hierarchical relationship. You must create the parent before you can create the child.

  For example, to create an ACL entry:

  1. The ACL must be created first by sending the JSON data of the ACL in the request body in a POST request to the URI of the `acls` collection:

     `/system/acls`

  2. The entry can then be created by sending the JSON data of the entry in the request body in a POST request to the URI of the ACL:

     `/system/acls/<ACL-name>/<ACL-type>/cfg_aces`

  Ports and interfaces do not have a parent that is clear from the resource hierarchy. The POST method JSON models for ports and interfaces include the field `referenced_by` to designate this relationship.

**More information**

Considerations for ports and interfaces on page 43
VSX peer switches and REST API access on page 35

# PUT method usage and considerations

The PUT method updates an instance of a resource by replacing the existing resource with the resource provided in the request body.

Configuration attributes that are set at the time a resource is created and that cannot be changed afterward are called **immutable** attributes. Configuration attributes that can be changed after a resource is created are called **mutable** attributes. The PUT method is used replace **mutable** attributes only.

- Not all resources support the PUT method. For information about the methods supported for a resource, see the ArubaOS-CX REST API Reference. The REST API must be in `read-write` mode to see all the PUT methods supported.
- The URI must specify a specific resource, not a collection.
- The URI must specify a resource that currently exists.
- For almost all resources, the PUT method is implemented as a strict replace operation.

All mutable configuration attributes are replaced. Any mutable attribute that the JSON data in request body does not include is either removed (if there is no default value) or reset to its default value.

### PUT request body requirements

The JSON data in the request body must include mutable (changeable) configuration attributes only.

The JSON model you must use for the PUT method request body is different from the JSON model used for the GET or the POST method.

The JSON model of a PUT method for a resource contains the mutable attributes only. In contrast, the JSON models for GET and POST methods can include both mutable and immutable attributes.

See the ArubaOS-CX REST API Reference for the JSON model used the PUT operation of a resource.

### PUT behavior

The PUT operation is a replace operation—not an update operation—because the resource instance in the request body replaces every changeable configuration attribute of the existing resource. Partial updates are not supported.

**CAUTION:** Any mutable attribute that the JSON data in request body does not include is either removed (if there is no default value) or reset to its default value.

For example:

- If you attempt a PUT operation on the system to change the host name, and you supply only the host name, you will destabilize the switch because the other attributes will be reset to their defaults, which might be empty.
- If you attempt to change the name of a VLAN and supply only the name in the PUT request, every other attribute in that VLAN is set back to its default of empty.

### Exceptions to the strict PUT behavior

For Network Analytics Engine agents, the PUT behavior is not a strict replace implementation. You can enable or disable agents without the supplying the entire set of configuration attributes in the PUT request body. For more information about the Network Analytic Engine resources, see the ArubaOS-CX Network Analytics Engine guide for your switch and software release.

### More information

Categories of resource attributes on page 15
VSX peer switches and REST API access on page 35

## Best practice method for building the PUT request body

Hewlett Packard Enterprise recommends the following procedure for building the PUT request body.

### Procedure

1. Use the GET method to obtain the response body for the resource you want to change.
2. Compare the response body from the GET request to the JSON model for the PUT request shown in the ArubaOS-CX REST API Reference, and remove all attribute value pairs that are not included in the PUT JSON model.

   The request body you send using the PUT method must match the JSON model shown for the PUT method—which is not the same as the JSON model for the POST or GET method. The JSON models for the GET and POST methods include both mutable and immutable attributes. The JSON model for the PUT method contains only the mutable attributes.

3. Change the values of the attributes to match your desired configuration.
4. Use the resulting JSON data as the request body for the PUT request.

# DELETE method usage and considerations

The DELETE method deletes an instance of a resource.

- Not all resources support the DELETE method. See the ArubaOS-CX REST API Reference for the methods supported by each resource. The REST API must be in `read-write` mode to see all the DELETE methods supported.
- The URI must specify a specific resource instance. The URI must not specify a collection.
- Child subcollections and resources are deleted when you delete the parent resource. For example, if you delete an ACL, its ACL entries are deleted automatically.
- DELETE requests do not contain a request body.
- DELETE requests do not return a response body.

The ArubaOS-CX real-time notifications subsystem enables external clients to connect to the switch through WebSocket secure and subscribe to receive real-time notifications about the switch resources and the configuration changes, state changes, and statistical information that interest them.

The Aruba OS-CX REST API, combined with the built-in databases that provide configuration, state, statistical data, and time-series data for the features and protocols running in the switch, provide a powerful means for switch programmability.

When a REST API is used for monitoring devices, it inherently implements the polling initiated from the client side. However, polling does not address the specific use cases in which network management systems need to receive live data or real-time events from the switch. There is a need to have a live notification subsystem that provides the remote network management system with real-time information about any changes that occur in the switch. Timely information about changes is important for troubleshooting and statistical data analyses, as well as for the immediate reaction to real-time events.

The ArubaOS-CX operating system provides built-in database that stores all configurations, states, and all statistical data for features and protocols running in the switch. When the switch image is built, a proprietary algorithm generates REST APIs and corresponding documentation for all resources in the database. Each resource or collection of the resources inside the switch is uniquely identified by its URI. The ArubaOS-CX real-time notification subsystem provides the ability for remote clients get real-time monitoring capability provided by the REST API by subscribing to the switch resource identified by its unique URI.

The WebSocket protocol is the considered to be the best protocol in the industry to use for real-time notifications. The WebSocket protocol was selected based on latency, throughput, resource utilization, network overhead, and security requirements. The handshake part of the WebSocket protocol uses HTTPS, so there is no need to open a new port on the switch side, and there is no need to provide a new authentication mechanism. Multiple clients and connections are supported.

ArubaOS-CX notification messages have JSON encoding, which was designed to align with REST payloads to allow clients to use combined REST and notification solutions.

The ability to subscribe to these push notifications about a variety of types of information about the switch, combined with the structured nature of the JSON data reported by the switch database, enable a form of network monitoring commonly called telemetry streaming.

Interested clients, known as subscribers, might include the following:

- Web clients such as the ArubaOS-CX Web UI
- Network management systems
- Monitoring scripts

# WebSocket secure connections for notifications

You subscribe to and receive notifications from the switch through a WebSocket secure (`wss://`) connection.

A WebSocket secure connection is a secure, persistent, and full-duplex connection between a client and a server. Either the client or the server can send data in the form of messages at any time.

The connection is established through a WebSocket handshake. A WebSocket handshake is an HTTP upgrade request to use to the WebSocket secure protocol, which is done by sending an upgrade header in the HTTPS request. To connect to the ArubaOS-CX switch, you must also pass the session cookie received from logging in to the REST API.

WebSocket connections to switches running ArubaOS-CX software remain active until the connection is closed, even after the session cookie expires.

For more information about the WebSocket protocol see *RFC 6455: The WebSocket Protocol* at:

**https://tools.ietf.org/html/rfc6455**

## Notification topics are switch resource URIs

When you subscribe to notifications, you subscribe to notifications about specific topics. A topic is the URI of a specific switch resource. That URI can contain a query string that specifies particular attributes of that resource.

For example, specifying the following URI as a topic results in notifications being sent when the administrative state or link state of any interface changes, but not when some other attribute of an interface changes:

```
/rest/v1/system/interfaces?attributes=admin_state,link_state
```

The ArubaOS-CX REST API Reference lists all the switch resources. You can use the GET method of the resource in the ArubaOS-CX REST API Reference to determine the URI for that switch resource, including the query string to specify an attribute or list of attributes.

## Rules for topic URIs

A topic is the URI of a switch resource:

- Not all switch resource URIs are supported as notification topics.

  The **Implementation Notes** section of the GET method of the resource in the ArubaOS-CX REST API Reference indicates if the resource is not supported by the notifications subsystem.
- Wildcard characters (`*`) are not supported.
- If Virtual Switching Extension (VSX) is enabled, you can specify a resource on a peer VSX switch by including `/vsx-peer` before the REST API and version identifier in the path portion of the URI.

  For example:

  ```
  /vsx-peer/rest/v1/system/bridge/vlans
  ```
- You can specify a specific resource instance or a collection of resources.

  Examples of specific resource instances:

  - `/rest/v1/system/vrfs/default`
  - `/rest/v1/system/bridge/vlans/DEFAULT_VLAN_1`
  - `/vsx-peer/rest/v1/system/bridge/vlans/DEFAULT_VLAN_1`

  Examples of resource collections:

  - `/rest/v1/system/vrfs/default/bgp_routers`
  - `/rest/v1/system/bridge/vlans`
- The `depth` query parameter is supported with a maximum value of `1` with resource collections only. For example:

- Correct: `/rest/v1/system/bridge/vlans?depth=1`
- Incorrect: `/rest/v1/system/bridge/vlans/2?depth=1`

- The `attributes` query parameter is supported. You can specify a comma-separated list of attribute names in the query string for either resource collections or resource instances. If attributes are specified, then the subscriber receives notification messages only when the value of one of the specified attributes changes.

  For example, the following URI specifies the administrative state and link state of all interfaces on the switch:

  ```
  /rest/v1/system/interfaces?attributes=admin_state,link_state
  ```

  The names of the attributes must match the names as documented in the ArubaOS-CX REST API Reference for the GET method of the resource.

## Notification security features

The notification feature uses secure WebSocket connections based on the TLS v1.2 protocol (Transport Layer Security version 1.2), which is the same protocol used for the REST HTTPS connections.

The switch uses self-signed certificates. To avoid certification errors, disable certificate verification when establishing the connection.

## ArubaOS-CX real-time notifications subsystem reference summary

The following information is intended as a quick reference for experienced users. Values are not configurable unless noted otherwise.

### Connection protocol

WebSocket secure (`wss://`)

### Port

443

### Message format

JSON

### Message types

The following are the supported message types:

- `subscribe`
- `unsubscribe`
- `success`
- `error`
- `notification`

### Authorization

Session cookie from successful HTTPS login request

### Notification resource URI

`wss://<IP-ADDR>/rest/v1/notification`

*<IP-ADDR>* is the IP address of the switch.

For example:

`wss://192.0.2.5/rest/v1/notification`

**Session idle timeout**

None

**Session hard timeout**

None

**Subscription persistence**

Subscriptions are active only while the WebSocket secure connection is open.

**Configuration maximums**

- Maximum number of subscribers per switch: 50
- Maximum number of subscriptions per subscriber: 80
- Maximum number of topics in one subscription message: eight

# Enabling the notifications subsystem on a switch

The ArubaOS-CX real-time notifications subsystem relies on the REST API, so the REST API must be enabled on the switch and VRF from which you want to receive notifications.

HTTPS server must be enabled on the specified VRF. The VRF you specify determines from which network the HTTPS server can be accessed. You can enable access on multiple VRFs, including user-defined VRFs.

**Procedure**

Enable REST API access on the VRF from which you will access the switch.

**More information**

Enabling access to the REST API on page 19
https-server vrf on page 83

# Establishing a WebSocket secure connection through a web browser

**Prerequisites**

- Access to the switch REST API must be enabled. The REST API access mode can be either read-only or read/write.
- The web browser you use must support WebSockets.

**Procedure**

1. Open a web browser page and log in to the switch Web UI or the REST API.

   The session cookie is managed by the browser and is shared among browser tabs.
2. From a different tab in the same browser, open the page that contains the WebSocket interface.

For example, many browsers have a plugin for WebSocket secure connections.

3. Connect to the switch at the following URL:

   `wss://<IP-ADDR>/rest/v1/notification`

   *<IP-ADDR>* is the IP address of the switch.

   For example:

   `wss://192.0.2.5/rest/v1/notification`

After the connection is established, you can use the interface to send subscribe or unsubscribe messages and to view the responses and notification messages.

**More information**

Example: Browser-based WebSocket connection on page 65

# Establishing a WebSocket secure connection using a script

**Prerequisites**

Access to the switch REST API must be enabled. The REST API access mode can be either read-only or read/write.

**Procedure**

- If you are using a script, you must include the actions to log in, get the session cookie, store the session cookie, and pass the session cookie with the WebSocket secure connection request.
- When you create the WebSocket secure connection, use the following URL:

  `wss://<IP-ADDR>/rest/v1/notification`

  *<IP-ADDR>* is the IP address of the switch.

  For example:

  `wss://192.0.2.5/rest/v1/notification`

- The exact methods to use to create connections and handle notification messages depend on the scripting language and library module you choose.

**More information**

Example: Python-based notification subscriber on page 61

# Subscribing to topics

**Prerequisites**

- You must have a WebSocket secure connection to the switch.
- Access to the switch REST API must be enabled. The REST API access mode can be either read-only or read/write.

**Procedure**

1. Using the WebSocket secure connection, send a subscribe message that contains the topics to which you want to subscribe and a poll interval hint, if any:

   For example:

```
{
  "type": "subscribe",
  "topics": [
    {
      "name": "/rest/v1/system/vrfs"
    },
    {
      "name": "/rest/v1/system/bridge/vlans/DEFAULT_VLAN_1?attributes=admin,oper_state_reason"
    }
  ],
  "hint": 5
}
```

   - If the subscriber already has a subscription to the specified topic, the following error is returned:

```
{
  "type":"error",
  "message":"The topic or combination of topics have been already subscribed."
}
```

   - If the URI in the topic name specifies a resource that is not in the configuration and state database, the following error is returned:

```
{"type":"error","message":"Object not found."}
```

Example of a message returned by a successful subscription attempt:

```
{
  "type": "success",
  "subscriber_name": "4bcf8uka90ki",
  "subscription_name": "ns83n58dky",
  "data": [
    {
      "topicname": "/rest/v1/system/bridge/vlans/DEFAULT_VLAN_1?attributes=admin,oper_state_reason",
      "resources": [
        {
          "operation": "",
          "uri": "/rest/v1/system/bridge/vlans/DEFAULT_VLAN_1",
          "values": {
            "admin": "up",
            "oper_state_reason": "no_member_port"
          }
        }
      ]
    },
    {
      "topicname": "/rest/v1/system/vrfs",
      "resources": [
        {
          "operation": "",
          "uri": "/rest/v1/system/vrfs/default",
          "values": {}
        },
        {
          "operation": "",
          "uri": "/rest/v1/system/vrfs/mgmt",
          "values": {}
        }
```

```
      ]
    }
  ]
}
```

**More information**

Parts of a subscribe message on page 57
Parts of a subscription success message on page 57

# Unsubscribing from topics

**Prerequisites**

- You must have a WebSocket secure connection to the switch.
- The switch must have REST API access enabled. The REST API access mode can be either read-only or read/write.

**Procedure**

Use the WebSocket secure connection to send an unsubscribe message that specifies the topic or topics from which you no longer want notifications.

Use a comma to separate topics in a list of topics.

You must be connected as the same subscriber that subscribed to the topic. For example, you must be using the same web browser connection or be passing the same session cookie with the request.

For example, to unsubscribe to notifications from the default VRF, send the following message through the WebSocket secure connection:

```
{
  "type": "unsubscribe",
  "topics": [
    {
      "name": "/rest/v1/system/vrfs/default"
    }
  ]
}
```

If the subscriber does not have a subscription to that topic, the following message is returned:

```
{
  "type": "error",
  "message": "No subscription for topic(s) <topic-name>."
}
```

The error can indicate that you have already unsubscribed, the connection was lost, or you attempted to unsubscribe from a different subscriber.

If the request is successful, the following message is returned:

```
{
  "type": "success",
  "message": "Successfully unsubscribe."
}
```

**More information**

Getting information about current subscribers and subscriptions on page 55

# Getting information about current subscribers and subscriptions

To get information about the subscribers receiving notifications from a switch, you must use the REST API.

Instructions and examples in this document use an IP address that is reserved for documentation, 192.0.2.5, as an example of the IP address for the switch. To access your switch, you must use the IP address or hostname of that switch.

### Prerequisites

You must be logged in to the switch REST API.

### Procedure

- To get the list of current subscribers, send a GET request to the `notification_subscribers` resource.

  For example:

  ```
  GET "https://192.0.2.5/rest/v1/system/notification_subscribers"
  ```

  The response body is a list of URIs. The identifier at the end of the URI string is the subscriber name.

  For example:

  ```
  [
    "rest/v1/system/notification_subscribers/z6901beisjgf",
    "rest/v1/system/notification_subscribers/18l9g87erb42"
  ]
  ```

- To get a list of all subscriptions of all subscribers, use the `depth=2` parameter when sending the GET request to the `notification_subscribers` resource.

  For example:

  ```
  GET "https://192.0.2.5/rest/v1/system/notification_subscribers?depth=2"
  ```

  The response body contains the list of subscriptions for each subscriber.

  In the following example,

  - Subscriber `z6901beisjgf` has two subscriptions:
    - `5mzo50lgoo`
    - `pouswxt9m9`
  - Subscriber `18l9g87erb42` has one subscription:
    - `dz95lljqwk`

  ```
  [
    {
      "name": "z6901beisjgf",
      "notification_subscriptions": {
        "5mzo50lgoo": "rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions/5mzo50lgoo",
        "pouswxt9m9": "rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions/pouswxt9m9"
      },
      "type": "ws"
    },
    {
      "name": "18l9g87erb42",
      "notification_subscriptions": {
        "dz95lljqwk": "rest/v1/system/notification_subscribers/18l9g87erb42/notification_subscriptions/dz95lljqwk"
      },
      "type": "ws"
  ```

```
  }
]
```

- To get detailed information about all subscriptions of all subscribers, use the `depth=2` parameter when sending the GET request to the `notification_subscribers` resource.

  For example:

  ```
  GET "https://192.0.2.5/rest/v1/system/notification_subscribers?depth=2"
  ```

- To get a list of subscriptions that belong to a specific subscriber, send a GET request to the `notification_subscriptions` resource of the subscriber.

  The following example gets the list of all the subscriptions of subscriber `z6901beisjgf`:

  ```
  GET "https://192.0.2.5/rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions"
  ```

  The response body is a list of URIs. The identifier at the end of the URI string is the subscription name.

  Example response body:

  ```
  [
    "rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions/5mzo50lgoo",
    "rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions/pouswxt9m9"
  ]
  ```

- To get detailed information about a specific subscription, send a GET request to the `notification_subscriptions/`*`{subscription-ID}`* resource for that subscription.

  The `notification_subscriptions` resource is a child resource of the specific subscriber:

  `/system/notification_subscribers/`*`{subscriber-id}`*`/notification_subscriptions/`*`{subscription-id}`*

  For example, to get information about subscription `5mzo50lgoo`, you must specify the subscriber name and the subscription name in the URI:

  ```
  GET "https://192.0.2.5/rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions/pouswxt9m9"
  ```

  Example response body:

  ```
  {
    "5mzo50lgoo": {
      "resource": [
        "/rest/v1/system/ports?attributes=admin,vlan_mode,vlan_tag,vlan_trunks,interfaces&depth=1"
      ]
    }
  }
  ```

- To get detailed information about all subscriptions of specific subscriber, use the `depth=1` parameter when sending the GET request to the `notification_subscriptions` resource of that subscriber.

  For example:

  ```
  GET "https://192.0.2.5/rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions?depth=2"
  ```

  Example response body:

  ```
  {
    "5mzo50lgoo": {
      "resource": [
        "/rest/v1/system/ports?attributes=admin,vlan_mode,vlan_tag,vlan_trunks,interfaces&depth=1"
      ]
    },
    "pouswxt9m9": {
      "resource": [
        "/rest/v1/system/interfaces?attributes=type,hw_intf_info,link_state,link_speed,error,other_config"
      ]
    }
  }
  ```

# Parts of a subscribe message

A subscribe message is the message sent when a subscriber requests a subscription to a topic on a switch. The subscribe message is in JSON format.

### Subscribe message example

```
{
  "type": "subscribe",
  "topics": [
    {
      "name": "/rest/v1/system/vrfs"
    },
    {
      "name": "/rest/v1/system/bridge/vlans/DEFAULT_VLAN_1?attributes=admin,oper_state_reason"
    }
  ],
  "hint": 5
}
```

### Components of a subscribe message

**`type`**

Required. For a subscribe message, you must specify the following value: `subscribe`

**`topics`**

Required. The value is a comma-separated list of one or more topics in JSON format. A topic includes one component:

**`name`**

Required. The name of the topic, identified by the URI of the switch resource, including the optional query string.

**`hint`**

Optional. Some resource attributes—typically in the statistics category—are not populated until a client requests the information. The value of `hint` specifies how often—in seconds—the notification subsystem is to request information about the topics in the list. The same `hint` value applies to all the topics in the list.

If the same resource is a topic in multiple subscriptions that have different values for `hint`, the notification subsystem uses the smallest value.

Default: 10

# Parts of a subscription success message

When a subscription request is successful, a subscription success message is returned. The subscription success message is in JSON format.

### Example success message

```
{
  "type": "success",
  "subscriber_name": "4bcf8uka90ki",
  "subscription_name": "ns83n58dky",
  "data": [
    {
      "topicname": "/rest/v1/system/bridge/vlans/DEFAULT_VLAN_1?attributes=admin,oper_state_reason",
      "resources": [
        {
          "operation": "",
          "uri": "/rest/v1/system/bridge/vlans/DEFAULT_VLAN_1",
          "values": {
```

```
            "admin": "up",
            "oper_state_reason": "no_member_port"
          }
        }
      ]
    },
    {
      "topicname": "/rest/v1/system/vrfs",
      "resources": [
        {
          "operation": "",
          "uri": "/rest/v1/system/vrfs/default",
          "values": {}
        },
        {
          "operation": "",
          "uri": "/rest/v1/system/vrfs/mgmt",
          "values": {}
        }
      ]
    }
  ]
}
```

### Components of subscription success message

**`type`**

Identifies the type of message. Success messages have the type: `success`

**`subscriber_name`**

Contains a unique identifier that represents the name of the subscriber.

**`subscription_name`**

Contains a unique identifier that represents the name of the specific subscription.

**`data`**

Contains a comma-separated list of one or more topics in JSON format.

### Components of a topic

In a subscription success message, each topic in the data contains the following components:

**`topicname`**

Contains the name of the topic, identified by the URI of the switch resource, including the optional query string.

**`resources`**

Contains a comma-separated list of one or more resources in JSON format. When the URI of a topic is a resource collection, a topic includes multiple resources. In the example message, the `vrfs` resource includes two VRF instances: `default` and `mgmt`.

Each resource includes the following components:

**`operation`**

The value of `operation` is empty for success messages. This component is used for notification messages only.

**`uri`**

Contains the URI of the resource instance within the resource collection. If the `topicname` is a resource instance instead of a collection, `uri` matches the path portion of the URI in `topicname`.

**values**

Contains the names and current values of the attributes that were specified in the query string of `topicname`.

# Parts of a notification message

A notification message is the message sent to the subscriber when there is a change to a switch resource that is the topic of a subscription. The notification message is in JSON format.

The content of a notification message depends on the type of change that occurred.

### Notification message examples

For the following examples, assume that the following subscribe message was used:

```
{
  "topics": [
    {
      "name": "/rest/v1/system/bridge/vlans?depth=1&attributes=name"
    }
  ],
  "type": "subscribe"
}
```

The subscriber receives a notification when the name of any VLAN changes:

- In the following example, VLAN7 has been added to the switch configuration:

```
{
  "data": [
    {
      "resources": [
        {
          "operation": "inserted",
          "uri": "/rest/v1/system/bridge/vlans/VLAN7",
          "values": {
            "name": "VLAN7"
          }
        }
      ],
      "topicname": "/rest/v1/system/bridge/vlans?depth=1&attributes=name"
    }
  ],
  "type": "notification"
}
```

- In the following example, VLAN7 has been deleted from the configuration:

```
{
  "data": [
    {
      "resources": [
        {
          "operation": "deleted",
          "uri": "/rest/v1/system/bridge/vlans/VLAN7",
          "values": {}
        }
      ],
      "topicname": "/rest/v1/system/bridge/vlans?depth=1&attributes=name"
```

```
      }
    ],
    "type": "notification"
  }
```

In the following example, the subscriber has subscribed to the following topic:

`/rest/v1/system/interfaces/1%2F1%2F2?attributes=name,admin_state`

If either the name or the administrative state of interface `1/1/2` changes, a notification message is sent. If attributes other than name or administrative state changes, no notification message is sent.

In the following example, the administrative state of the interface changed to `up`.

```
{
  "data": [
    {
      "resources": [
        {
          "operation": "modified",
          "uri": "/rest/v1/system/interfaces/1%2F1%2F2",
          "values": {
            "admin_state": "up"
          }
        }
      ],
      "topicname": "/rest/v1/system/interfaces/1%2F1%2F2?attributes=name,admin_state"
    }
  ],
  "type": "notification"
}
```

### Components of a notification message

**`type`**

Identifies the type of message. Notification messages have the type: `notification`

**`data`**

Contains a comma-separated list of one or more topics in JSON format.

### Components of a topic

In a notification message, each topic in the data contains the following components:

**`topicname`**

Contains the name of the topic, identified by the URI of the switch resource, including the optional query string.

**`resources`**

Contains a comma-separated list of one or more resources in JSON format. When the URI of a topic is a resource collection, a topic includes multiple resources.

Each resource includes the following components:

**`operation`**

For notification messages, operation is one of the following values:

**`inserted`**

The resource or resource attribute was added to the configuration of the switch.

**deleted**

The resource or resource attribute was deleted from the switch.

**modified**

The resource or resource attribute changed.

**uri**

Contains the URI of the resource instance within the resource collection. If the `topicname` is a resource instance instead of a collection, `uri` matches the path portion of the URI in `topicname`.

**values**

The content of `values` depends on the operation:

- When the `operation` value is `deleted`, `values` is empty.
- When the `operation` value is `inserted`, `values` contains the current names and values of the attributes specified in the query portion of the `topicname`. If no query string was included in `topicname`, all attributes and values for that resource are included.
- When the `operation` value is `modified`, `values` contains the name and current value of the attribute in the query string that changed value:
  - If no query string was included in `topicname`, all attributes and values for that resource are included.
  - If multiple attributes are included in the query string of a topic and only some of those attribute values changed, only the changed attributes are included.
  - If an attribute that was not included in the query string changes, no notification message is sent because that attribute is not part of the subscription.

# Example: Python-based notification subscriber

### About the example

The following example, `websocket-client-interfaces.py`, is a Python script that creates an interactive client that subscribes to notifications about interfaces:

- Python 3.4 or later is required.
- The script uses the Python 3 versions of the Tornado module to do create and handle the HTTP requests and WebSocket secure connection. For more information about Tornado, see:

  **http://www.tornadoweb.org/**
- The script uses the Python 3 versions of the requests module to send REST API requests to the switch. For more information about the requests module, see:

  **http://docs.python-requests.org/**
- Access to the switch REST API must be enabled on the VRF through which this script will connect to the switch.

In the script, notice the following:

- The value of user name and password used in the login routine is set in the USER and PASSWORD parameters. Hewlett Packard Enterprise recommends that you modify the script to collect the user name and password interactively, and pass them as data so they do not show in the system logs in cleartext.
- The session cookie is created as part of the `login` routine and is stored in `cookie_header`.
- The WebSocket secure connection is made by the `connect` routine calling the Tornado `websocket_connect` function. The session cookie, `cookie_header`, is passed as part of the `http_request` created by the `HTTPRequest` function.
- The list of topics is collected from the input by the following routine:

```
def collect_topics(args):
    topics_list = []
    if len(args) > 2:
        length = len(args)
        for i in range(2, length):
            topics_list.append(args[i])
    return topics_list
```

- The subscribe message is created in the following routine:

```
    def create_json_dict(self, topics_list):
        json_dict = dict()
        json_dict["type"] = "subscribe"
        topic_list = []
        for i in range(len(topics_list)):
            topic_dict = dict()
            topic_dict["name"] = topics_list[i]
            topic_list.append(topic_dict)
        json_dict["topics"] = topic_list
        return json_dict
```

- Received messages are handled in the lines that begin with the following:

```
    def run(self, topics_list):
```

### Example Python script

```
# This script is executed as - python websocket-client.py {switchNotificatonURL} {topicURI(s)}
# Example of executing the script with multiple topics:
# python websocket-client.py
# "wss://{ipAddress}:{port}/rest/v1/notification"
# "/rest/v1/system/interfaces/1%2F1%2F1"
# "/rest/v1/system/interfaces/1%2F1%2F2"
#
#
from requests import post
from tornado import escape
import json
from tornado.ioloop import IOLoop, PeriodicCallback
from tornado import gen
from tornado.websocket import websocket_connect
from tornado.httpclient import HTTPRequest
import sys
import traceback
from ssl import PROTOCOL_SSLv23

USER = 'admin3'
PASSWORD = 'admin3F@rM#'
```

```
PROXY_DICT = {'http': None, 'https': None}
REQUEST_TIMEOUT = 50
CONNECT_TIMEOUT = 50

class Client(object):
    def __init__(self, url, timeout, topics_list):
        self.url = url
        self.timeout = timeout
        self.ioloop = IOLoop.instance()
        self.ws = None
        self.cookie_header = self.login()
        self.count = 0
        self.connect(url, self.cookie_header, topics_list)
        self.ioloop.start()

    @gen.coroutine
    def connect(self, ws_uri, cookie_header, topics_list):
        print("trying to connect")
        try:
            http_request = HTTPRequest(url=ws_uri,
                                       headers=cookie_header,
                                       follow_redirects=True,
                                       ssl_options={"ssl_version":
                                                    PROTOCOL_SSLv23},
                                       validate_cert=False)
            self.ws = yield websocket_connect(http_request)
        except Exception as e:
            print("connection error" + str(e))
        else:
            print("connected")
            self.run(topics_list)

    @gen.coroutine
    def run(self, topics_list):
        json_dict = self.create_json_dict(topics_list)
        self.ws.write_message(escape.utf8(json.dumps(json_dict)))
        while True:
            msg = yield self.ws.read_message()
            self.count = self.count + 1
            print(msg)
            if self.count == 1:
                msg_in_json = self.check_if_JSON(msg)
                if msg_in_json is not None:
                    success_test = self.check_if_success(msg_in_json)
                    if success_test:
                        print("PASS - Initial return JSON")
                    else:
                        print("FAIL - Initial return JSON")
            if msg is None:
                print("connection closed")
                self.ws = None
                break

    def check_if_JSON(self, result):
        try:
            msg_json = json.loads(result)
        except ValueError:
            print("The message received is not a valid JSON")
        return msg_json

    def check_if_success(self, json_response):
        pass_type = pass_resource = False
        if "type" in json_response:
```

```
            type_msg = json_response["type"]
            if type_msg == "success":
                pass_type = True
        if "data" in json_response:
            for each in json_response['data']:
                if "resources" in each:
                    pass_resource = True
        return pass_type and pass_resource

    def login(self, username=None, password=None, proxies=PROXY_DICT):
        if username is not None:
            assert password is not None, "Must provide password for Login"
        if not username:
            username = USER
        if not password:
            password = PASSWORD

        params = {'username': username,
                  'password': password}
        login_url = NOTIFICATION_URL.replace("wss", "https")
        login_url = login_url.replace("notification", "login")
        login_header = {'Content-Type': 'application/x-ww-form-urlencoded'}
        response = post(login_url, verify=False, headers=login_header,
                        params=params, proxies=proxies)
        cookie_header = {'Cookie': response.headers['set-cookie']}
        return cookie_header

    def create_json_dict(self, topics_list):
        json_dict = dict()
        json_dict["type"] = "subscribe"
        topic_list = []
        for i in range(len(topics_list)):
            topic_dict = dict()
            topic_dict["name"] = topics_list[i]
            topic_list.append(topic_dict)
        json_dict["topics"] = topic_list
        return json_dict


def collect_topics(args):
    topics_list = []
    if len(args) > 2:
        length = len(args)
        for i in range(2, length):
            topics_list.append(args[i])
    return topics_list


if __name__ == "__main__":
    try:
        NOTIFICATION_URL = sys.argv[1]
        topics = collect_topics(sys.argv)
        client = Client(NOTIFICATION_URL, 10,  topics)
    except KeyboardInterrupt:
        print("Shutdown requested...exiting")
    except Exception:
        traceback.print_exc(file=sys.stdout)
    sys.exit(0)
```

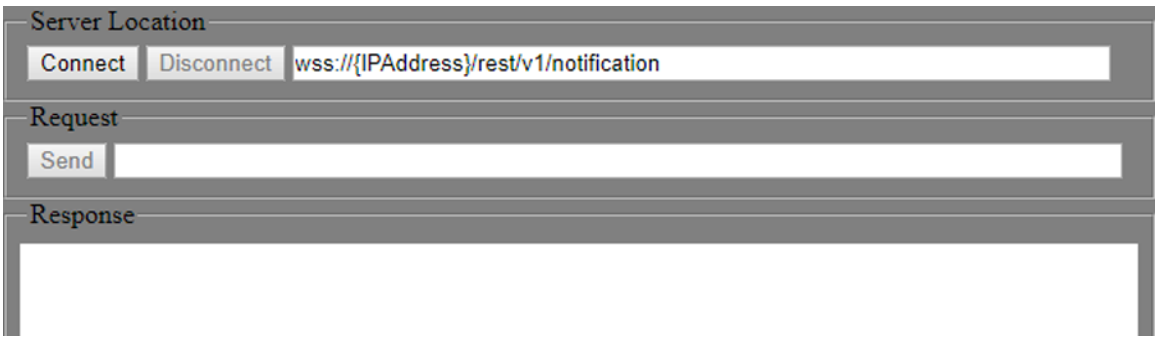
# Example: Browser-based WebSocket connection

### About the example

The following example, `websocket-client.html`, uses HTML and Javascript to create webpage that you can use to establish a WSS connection and send and receive notification messages.

- Access to the switch REST API must be enabled on the VRF through which this browser will connect to the switch.
- Before you can use the HTML page, you must log in to the switch Web UI or REST API from a separate tab in the same web browser session. The browser shares the session cookie between tabs.
- When the browser page is open, in **Server Location**, substitute the switch IP address for `{IPAddress}` in `wss://{IPAddress}/rest/v1/notification`, then click **Connect**.
- Enter the subscription message in **Request** and click **Send**.
- Responses and notifications are shown in **Response**.

### Example screen




### Example HTML source

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Web Socket Client Example</title>
    <script type="text/javascript">
        window.onload = function () {
            var conn;
            var log = document.getElementById("log");
            var msg = document.getElementById("msg");

            function appendLog(item) {
                var doScroll = log.scrollTop === log.scrollHeight - log.clientHeight;
                log.appendChild(item);
                if (doScroll) {
                    log.scrollTop = log.scrollHeight - log.clientHeight;
                }
            }

            document.getElementById("connect").onclick = function () {
                var server = document.getElementById("wsURL");
                conn = new WebSocket(server.value);
                if (window["WebSocket"]) {
                    if (conn) {
                        conn.onopen = function (evt) {
                            document.getElementById("disconnect").disabled = false
```

```
                        document.getElementById("sendMsg").disabled = false
                        document.getElementById("connect").disabled = true
                        document.getElementById("status").innerHTML = "Connection opened"
                    }
                    conn.onclose = function (evt) {
                        document.getElementById("status").innerHTML = "Connection closed"
                        document.getElementById("connect").disabled = false
                    };
                    conn.onmessage = function (evt) {
                        var messages = evt.data.split('\n');
                        for (var i = 0; i < messages.length; i++) {
                            var item = document.createElement("pre");
                            item.innerText = messages[i];
                            appendLog(item);
                        }
                    }
                }
            } else {
                var item = document.createElement("pre");
                item.innerHTML = "<b>Your browser does not support WebSockets.</b>";
                appendLog(item);
            }
        };

        document.getElementById("disconnect").onclick = function () {
            conn.close()
            document.getElementById("sendMsg").disabled = true
            document.getElementById("connect").disabled = false
            document.getElementById("disconnect").disabled = true
            document.getElementById("status").innerHTML = "Connection closed"
        };

        document.getElementById("form").onsubmit = function () {
            if (!conn) {
                return false;
            }
            if (!msg.value) {
                return false;
            }
            conn.send(msg.value);
            var item = document.createElement("pre");
            item.classList.add("subscribeMsg");
            item.innerHTML = msg.value;
            appendLog(item);
            return false;
        };
    };

</script>
<style type="text/css">
    html {
        overflow: hidden;
    }

    body {
        overflow: hidden;
        padding: 0;
        margin: 0;
        width: 100%;
        height: 100%;
        background: gray;
    }

    #log {
```

```
            background: white;
            margin: 0;
            padding: 0.5em 0.5em 0.5em 0.5em;
            top: 1.5em;
            left: 0.5em;
            right: 0.5em;
            bottom: 3em;
            overflow: auto;
            position: absolute;
            height: 530px;
        }

        #form {
            padding: 0 0.5em 0 0.5em;
            margin: 0;
            position: absolute;
            bottom: 3em;
            top: 5em;
            left: 8px;
            width: 100%;
            overflow: hidden;
        }

        #serverLocation {
            padding-top: 0.3em;
        }

        #requestSection {
            height: 38px;
        }

        #responseMsgSection {
            height: 570px;
            position: relative;
        }
    </style>
</head>
<body>
<fieldset id="serverLocation">
    <legend>Server Location</legend>
    <div>
        <input type="button" id="connect" value="Connect"/>
        <input type="button" id="disconnect" value="Disconnect" disabled/>
        <input type="text" id="wsURL" value="wss://{IPAddress}/rest/v1/notification" size="64">
        <span id="status"></span>
    </div>
</fieldset>
<fieldset id="requestSection">
    <legend>Request</legend>
    <form id="form">
        <input id="sendMsg" type="submit" value="Send" ; disabled/>
        <input type="text" id="msg" size="80"/>
    </form>
</fieldset>
<fieldset id="responseMsgSection">
    <legend>Response</legend>
    <div id="log"></div>
</fieldset>
</body>
</html>
```

# Example: Getting information about notification subscriptions

### Getting a list of subscribers

The following example gets the list of all subscribers that are subscribed to notifications from a switch resource:

```
GET "https://192.0.2.5/rest/v1/system/notification_subscribers"
```

Example response body:

```
[
  "rest/v1/system/notification_subscribers/z6901beisjgf",
  "rest/v1/system/notification_subscribers/18l9g87erb42"
]
```

The response body is a list of URIs. The identifier at the end of the URI string is the subscriber name. You can use that identifier to get information about the subscriptions for that subscriber.

You can get more information about all the subscribers by using the `depth` parameter. For example, you can specify `depth=1` to get a list of the subscriptions for each subscriber:

```
GET "https://192.0.2.5/rest/v1/system/notification_subscribers?depth=1"
```

Example response body:

```
[
  {
    "name": "z6901beisjgf",
    "notification_subscriptions": {
      "5mzo50lgoo": "rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions/5mzo50lgoo",
      "pouswxt9m9": "rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions/pouswxt9m9"
    },
    "type": "ws"
  },
  {
    "name": "18l9g87erb42",
    "notification_subscriptions": {
      "dz95lljqwk": "rest/v1/system/notification_subscribers/18l9g87erb42/notification_subscriptions/dz95lljqwk"
    },
    "type": "ws"
  }
]
```

You can specify `depth=2` to further expand the list of subscription URIs for all the subscribers, or you can use the `notifications_subscriptions` resource to get information about subscriptions by subscription name.

### Getting information about subscriptions

The following example gets the list of all the subscriptions of subscriber `z6901beisjgf`:

```
GET "https://192.0.2.5/rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions"
```

Example response body:

```
[
  "rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions/5mzo50lgoo",
  "rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions/pouswxt9m9"
]
```

The response body is a list of URIs. The identifier at the end of the URI string is the subscription name.

You can specify `depth=1` to expand the URI of the subscriptions to get more information about all the subscriptions. For example:

```
GET "https://192.0.2.5/rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions?depth=1"
```

Example response body:

```
{
  "5mzo50lgoo": {
    "resource": [
      "/rest/v1/system/ports?attributes=admin,vlan_mode,vlan_tag,vlan_trunks,interfaces&depth=1"
    ]
  },
  "pouswxt9m9": {
    "resource": [
      "/rest/v1/system/interfaces?attributes=type,hw_intf_info,link_state,link_speed,error,other_config"
    ]
  }
}
```

You can also get information about a subscription by the subscription name for a specific subscriber. For example:

```
GET "https://192.0.2.5/rest/v1/system/notification_subscribers/z6901beisjgf/notification_subscriptions/pouswxt9m9"
```

Example response body:

```
{
  "5mzo50lgoo": {
    "resource": [
      "/rest/v1/system/ports?attributes=admin,vlan_mode,vlan_tag,vlan_trunks,interfaces&depth=1"
    ]
  }
}
```

# Examples: GET method

Instructions and examples in this document use an IP address that is reserved for documentation, 192.0.2.5, as an example of the IP address for the switch. To access your switch, you must use the IP address or hostname of that switch.

- Get the list of all VLANS:

  ```
  GET "https://192.0.2.5/rest/v1/system/bridge/vlans"
  ```

- Expand the list of URIs in the `vlans` collection by one level, which replaces the URI for the VLAN with the JSON data for that VLAN.

  ```
  GET "https://192.0.2.5/rest/v1/system/bridge/vlans?depth=1"
  ```

- Use the `count` parameter to get the total number of VLANs:

  ```
  GET "https://192.0.2.5/rest/v1/system/bridge/vlans?count"
  ```

- Use the `count` parameter with the `filter` parameter to get the total number of interfaces in a `down` administrative state:

  ```
  GET "https://192.0.2.5/rest/v1/system/interfaces?count=true&filter=admin_state:down"
  ```

- Use the `filter` parameter with the value `type:static` to get a list of only the static VLANs:

  ```
  GET "https://192.0.2.5/rest/v1/system/bridge/vlans?filter=type:static"
  ```

- Use the `filter` parameter to get the BGP routes that have 1.1.1.1 as a peer:

  ```
  GET "https://192.0.2.5/rest/v1/system/vrfs/default/bgp_routes?filter=peer:1.1.1.1"
  ```

- Use the `attributes` parameter to get all ports but show only the attributes `name` and `ipv4_address`:

  ```
  GET "https://192.0.2.5/rest/v1/system/ports?attributes=name,ipv4_adress"
  ```

- Use the wildcard character to get a list of routes for all VRFs.

  ```
  GET "https://192.0.2.5/rest/v1/system/vrfs/*/routes"
  ```

- Use the `selector` parameter to get all the configuration attributes of VLAN 100:

  ```
  GET "https://192.0.2.5/rest/v1/system/bridge/vlans/100?selector=configuration"
  ```

- Use the `selector` parameter to get all the system attributes that are in the categories configuration and status:

  ```
  GET "https://192.0.2.5/rest/v1/system?selector=category,status"
  ```

# Example: Configuration management using REST APIs

**Downloading a configuration image**

Example of downloading the current configuration:

```
GET "https://192.0.2.5/rest/v1/fullconfigs/running-config"
```

Example of downloading the startup configuration:

```
GET "https://192.0.2.5/rest/v1/fullconfigs/startup-config"
```

On successful completion, the switch returns response code 200 OK and a response body containing the entire configuration in JSON format.

### Uploading a configuration image

The following example shows the curl command to upload a configuration to become the running configuration. The configuration being uploaded—represented as ellipsis but not shown in this example—is in JSON format in the body of the command (enclosed in braces).

The running configuration is the only configuration that can be updated using this technique. However you can copy configurations (see Copying configurations).

```
$ curl -k -X PUT \
-b /tmp/auth_cookie \
"https://192.0.2.5/rest/v1/fullconfigs/running-config" \
-d '
{
…
}'
```

On successful completion, the switch returns response code 200 OK.

### Copying configurations

To replace an existing configuration with another, use a REST PUT request to the destination configuration, using the `from` query string parameter to specify the source configuration.

- The source or the destination configuration must be either `running-config` or `startup-config`. You cannot copy a checkpoint to a different checkpoint.
- If you specify a destination checkpoint that exists, an error is returned. You cannot overwrite an existing checkpoint.

The syntax is as follows:

```
PUT "https://<IPADDR>/rest/v1/fullconfigs/<destination-config>?
from=/rest/v1/fullconfigs/<source-config>"
```

Example of copying the running configuration to the startup configuration:

```
$ curl -k -X PUT \
-b /tmp/auth_cookie -D-
"https://192.0.2.5/rest/v1/fullconfigs/startup-config?
from=/rest/v1/fullconfigs/running-config"
```

Example of copying the startup configuration to the running configuration:

```
$ curl -k -X PUT \
-b /tmp/auth_cookie -D-
"https://192.0.2.5/rest/v1/fullconfigs/running-config?
from=/rest/v1/fullconfigs/startup-config"
```

Example of copying a checkpoint to the running configuration:

```
$ curl -k -X PUT \
-b /tmp/auth_cookie -D-
"https://192.0.2.5/rest/v1/fullconfigs/running-config?
from=/rest/v1/fullconfigs/MyCheckpoint"
```

Example of copying the running configuration to a checkpoint:

```
$ curl -k -X PUT \
-b /tmp/auth_cookie -D-
"https://192.0.2.5/rest/v1/fullconfigs/MyCheckpoint?
from=/rest/v1/fullconfigs/running-config"
```

# Example: Firmware upgrade using REST APIs

### Uploading a file to the secondary firmware image

In the following example, a curl command is used to upload the firmware image file from the local workstation to the switch and post that file as the secondary firmware image. The `-F` option specifies that the POST method is used to upload the file.

```
$ curl -k -b /tmp/auth_cookie \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
-F "fileupload=@/myfirmwarefiles/myswitchfirmware_2017020905.swi" \
https://192.0.2.5/rest/v1/firmware?image=secondary
```

### Booting the system using the secondary firmware image

Method and URL:

```
POST "https://192.0.2.5/rest/v1/boot?image=secondary"
```

Example curl command:

```
$ curl -k POST -b /tmp/auth_cookie \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
"https://192.0.2.5/rest/v1/boot?image=secondary"
```

# Example: Log operations using REST APIs

### Event logs

A GET request to `/rest/v1/logs/event` URI returns all entries from all the event logs on the switch, including logs from internal processes.

The information returned by this request was not optimized for human readability. If you want to examine the log entries, Hewlett Packard Enterprise recommends that you use the Web UI. The Web UI also provides a method to export log entries.

In the following example, the MESSAGE_ID parameter filters the output to include event log messages only:

- MESSAGE_ID `50c0fa81c2a545ec982a54293f1b1945` identifies event log messages from the active management module.
- MESSAGE_ID `73d7a43eaf714f97bbdf2b251b21cade` identifies event log messages from the standby management module. Not all switches have a standby management module.

Example method and URI:

```
GET "https://192.0.2.5/rest/v1/logs/event?
limit=1000&
priority=4&
since=24%20hour%20ago&
MESSAGE_ID=50c0fa81c2a545ec982a54293f1b1945,73d7a43eaf714f97bbdf2b251b21cade"
```

### Audit logs

A GET request to `/rest/v1/logs/audit` URI returns all entries from the audit logs on the switch.

For a list of supported query parameters, see the ArubaOS-CX REST API Reference

Example method and URI:

```
GET "https://192.0.2.5/rest/v1/logs/audit?
since=24%20hour%20ago&
usergroup=administrators&
session=CLI"
```

# Example: Ping operations using REST APIs

This example gets ping statistics for the ping target.

Example method and URI:

```
GET "https://192.168.0.1/rest/v1/ping?
ping_target=192.0.2.10&
is_ipv4=true&
ping_data_size=100&
ping_time_out=2&
ping_repetitions=1&
ping_type_of_service=0&
include_time_stamp=false&
include_time_stamp_address=false&
record_route=false&
mgmt=false"
```

On successful completion, the switch returns response code 200 OK and a response body containing the output string produced by the ping operation.

# Example: Traceroute operations using REST APIs

Example method and URI:

```
GET "https://192.0.2.5/rest/v1/traceroute?
ip=192.0.2.10&
is_ipv4=true&
timeout=3&
destination_port=33434&
max_ttl=30&
min_ttl=1&
probes=3&
mgmt=false"
```

On successful completion, the switch returns response code 200 OK and a response body containing the output string produced by the traceroute operation.

# Example: User management using REST APIs

### Creating a user

Method and URI:

```
POST "https://192.0.2.5/rest/v1/system/users"
```

Request body:

```
{
  "name": "myadmin",
```

```
  "password": "P@sswr0d",
  "user_group": "/rest/v1/system/user_groups/administrators",
  "origin": "configuration"
}
```

Example curl command:

```
$ curl -k POST \
-b /tmp/auth_cookie \
"https://192.0.2.5/rest/v1/system/users" -d '
{
  "name": "myadmin",
  "password": "P@sswr0d",
  "user_group": "/rest/v1/system/user_groups/administrators",
  "origin": "configuration"
}'
```

On successful completion, the switch returns response code 201 Created.

### Changing a password

Method and URI:

```
PUT "https://192.0.2.5/rest/v1/system/users/myadmin"
```

Request body:

```
{
  "password": "P@sswr0d2g",
}
```

Example curl command:

```
$ curl -k -X PUT \
-b /tmp/auth_cookie \
"https://192.0.2.5/rest/v1/system/users/myadmin" -d '
{
  "password": "P@sswr0d2g",
}'
```

On successful completion, the switch returns response code 200 OK.

### Deleting a user

Method and URI:

```
DELETE "https://192.0.2.5/rest/v1/system/users/myadmin"
```

Example curl command:

```
$ curl -k -X DELETE \
-b /tmp/auth_cookie \
"https://192.0.2.5/rest/v1/system/users/myadmin"
```

On successful completion, the switch returns response code 204 No Content.

# Example: Creating an ACL with a port using REST APIs

This example shows creating the following ACL and port configuration on a switch at IP address 192.0.2.5:

```
interface 1/1/2
    no shutdown
```

```
    apply access-list ip ACLv4 out
access-list ip ACLv4
    10 permit tcp 10.0.100.101 eq 80 10.0.100.102 eq 8000
```

1. Creating the ACL.

```
$ curl -k --noproxy 192.0.2.5 POST \
-b /tmp/auth_cookie -d '{
"cfg_version": 0,
"list_type": "ipv4",
"name": "ACLv4"}'
"https://192.0.2.5/rest/v1/system/acls"
```

2. Creating an ACL entry.

```
$ curl -k --noproxy 192.0.2.5 POST \
-b /tmp/auth_cookie -d '{
"action": "permit",
"dst_ip": "10.0.100.102/255.255.255.255",
"dst_l4_port_max": 8000,
"dst_l4_port_min": 8000,
"protocol": 6,
"sequence_number": 10,
"src_ip": "10.0.100.101/255.255.255.255",
"src_l4_port_max": 80,
"src_l4_port_min": 80}'
"https://192.0.2.5/rest/v1/system/acls/ACLv4/ipv4/cfg_aces"
```

3. Getting the ACL configuration information to use in the next step. Ellipses (…) represent data not shown in the example.

```
$ curl -k --noproxy 192.0.2.5 GET \
-b /tmp/auth_cookie \
"https://192.0.2.5/rest/v1/system/acls/ACLv4/ipv4?selector=configuration"
{
…
  "cfg_aces": {},
  "cfg_version": 0
…
  "list_type": "ipv4",
  "name": "ACLv4"
…
}
```

4. Updating the ACL configuration using the return body received from the GET request performed in the previous step.

   When you send a PUT request, the JSON request body must not contain immutable attributes. The ArubaOS-CX REST API Reference model for the PUT method of the resource shows the mutable attributes. Any mutable attributes you do not include in the PUT request body are set to their defaults, which could be empty.

   The ArubaOS-CX REST API Reference JSON model for the PUT method of the `/system/acls/{id1}/{id2}` resource shows the following example:

```
{
  "cfg_aces": {
    "integer": "URL"
  },
  "cfg_version": 0
}
```

   The following example shows the request to update the ACL configuration:

```
$ curl -k --noproxy 192.0.2.5 -X PUT \
-b /tmp/auth_cookie -d '{
"cfg_aces":{"10":"/rest/v1/system/acls/ACLv4/ipv4/cfg_aces/10"},
"cfg_version":1}' \
"https://192.0.2.5/rest/v1/system/acls/ACLv4/ipv4"
```

5. Creating port 1/1/2.

```
$ curl -k --noproxy 192.0.2.5 POST \
-b /tmp/auth_cookie -d '{
"name": "1/1/2",
"admin":"up",
"interfaces":["/rest/v1/system/interfaces/1%2F1%2F2"],
"referenced_by":"/rest/v1/system/vrfs/default"
"description":"referenced by default vrf"}' \
"https://192.0.2.5/rest/v1/system/ports"
```

6. Getting the configuration information for the interface.

   The GET response body includes only the configuration attributes that have been set.

```
$ curl -k --noproxy 192.0.2.5 GET \
-b /tmp/auth_cookie \
"https://192.0.2.5/rest/v1/system/interfaces/1%2F1%2F2?selector=configuration"
{
  "options": {},
  "other_config": {},
  "udld_arubaos_compatibility_mode": "forward_then_verify",
  "udld_compatibility": "aruba_os",
  "udld_enable": false,
  "udld_interval": 7000,
  "udld_retries": 4,
  "udld_rfc5171_compatibility_mode": "normal",
  "user_config": {}
}
```

7. Verifying which configuration attributes are mutable and therefore can be included in the PUT request.

   When you send a PUT request, the JSON request body must not contain immutable attributes. The ArubaOS-CX REST API Reference JSON model for the PUT method of the resource shows the mutable attributes. Any mutable attributes you do not include in the PUT request body are set to their defaults, which could be empty.

   The ArubaOS-CX REST API Reference JSON model for the PUT method of the `/system/interfaces/{id}` resource shows the following example:

```
{
  "description": "string",
  "options": {},
  "other_config": {},
  "selftest_disable": true,
  "udld_arubaos_compatibility_mode": "string",
  "udld_compatibility": "string",
  "udld_enable": true,
  "udld_interval": 0,
  "udld_retries": 0,
  "udld_rfc5171_compatibility_mode": "string",
  "user_config": {}
}
```

8. Enabling the interface using all the attributes in the return body received from the GET request, modifying the `user_config` attribute to be: `"user_config":{"admin":"up"}`

```
$ curl -k --noproxy 192.0.2.5 -X PUT \
-b /tmp/auth_cookie -d '{
  "options": {},
  "other_config": {},
  "udld_arubaos_compatibility_mode": "forward_then_verify",
  "udld_compatibility": "aruba_os",
  "udld_enable": false,
  "udld_interval": 7000,
  "udld_retries": 4,
  "udld_rfc5171_compatibility_mode": "normal",
  "user_config": {
    "admin": "up"
  }
}' \
"https://192.0.2.5/rest/v1/system/interfaces/1%2F1%2F2"
```

In the preceding example, the following mutable attributes listed in the previous step were not included, so they are set to their default values, which could be empty:

- `description`
- `selftest_disable`

9. Getting the port configuration information to use in the next step.

Ellipses (…) represent data not shown in the example.

```
$ curl -k --noproxy 192.0.2.5 GET \
-b /tmp/auth_cookie \
"https://192.0.2.5/rest/v1/system/ports/1%2F1%2F2?selector=configuration"
{
  "aclv4_out_cfg": {},
  "aclv4_out_cfg_version": {},
  "admin": {},
  "arp_timeout": 1800,
…
  },
…
  "virtual_ip4_routers": {},
  "virtual_ip6_routers": {},
  "vlan_trunks": []
}
```

10. Adding the ACL information to the port using the return body received from the GET request performed in the previous step after verifying the values that are permitted in the JSON model for the PUT method. The modified values are shown in the following example.

Ellipses (…) represent data not shown in the example.

```
$ curl -k --noproxy 192.0.2.5 -X PUT \
-b /tmp/auth_cookie -d '{
…
"admin":"up",
"interfaces":["/rest/v1/system/interfaces/1%2F1%2F2"],
"aclv4_out_cfg":"/rest/v1/system/acls/ACLv4/ipv4",
"aclv4_out_cfg_version":0,
…
```

```
}' -D- \
"https://192.0.2.5/rest/v1/system/ports/1%2F1%2F2"
```

# Example: Creating a VLAN with a port using REST APIs

This example shows creating the following VLAN and port configuration on a switch at IP address 192.0.2.5:

```
vlan 2
    no shutdown
interface 1/1/2
    no shutdown
    no routing
    vlan access 2
```

1. Creating the VLAN.

```
$ curl -k --noproxy 192.0.2.5 POST \
-b /tmp/auth_cookie -d '{
"name":"VLAN2",
"id":2,
"type":"static",
"admin":"up"}' \
"https://192.0.2.5/rest/v1/system/bridge/vlans"
```

2. Creating a port and configure the VLAN information.

```
$ curl -k --noproxy 192.0.2.5 POST \
-b /tmp/auth_cookie -d '{
"name": "1/1/2",
"admin":"up",
"interfaces":["/rest/v1/system/interfaces/1%2F1%2F2"],
"vlan_mode":"access",
"vlan_tag":"/rest/v1/system/bridge/vlans/2",
"referenced_by":"/rest/v1/system/bridge"
"description":"referenced by /system/bridge"}' \
-D- "https://192.0.2.5/rest/v1/system/ports"
```

3. Getting the configuration information for the interface.

   The GET response body includes only the configuration attributes that have been set.

```
$ curl -k --noproxy 192.0.2.5 GET \
-b /tmp/auth_cookie \
"https://192.0.2.5/rest/v1/system/interfaces/1%2F1%2F2?selector=configuration"
{
  "options": {},
  "other_config": {},
  "udld_arubaos_compatibility_mode": "forward_then_verify",
  "udld_compatibility": "aruba_os",
  "udld_enable": false,
  "udld_interval": 7000,
  "udld_retries": 4,
  "udld_rfc5171_compatibility_mode": "normal",
  "user_config": {}
}
```

4. Verifying which configuration attributes are mutable and therefore can be included in the PUT request.

   When you send a PUT request, the JSON request body must not contain immutable attributes. The ArubaOS-CX REST API Reference JSON model for the PUT method of the resource shows the mutable attributes. Any mutable attributes you do not include in the PUT request body are set to their defaults, which could be empty.

The ArubaOS-CX REST API Reference JSON model for the PUT method of the `/system/interfaces/{id}` resource shows the following example:

```
{
  "description": "string",
  "options": {},
  "other_config": {},
  "selftest_disable": true,
  "udld_arubaos_compatibility_mode": "string",
  "udld_compatibility": "string",
  "udld_enable": true,
  "udld_interval": 0,
  "udld_retries": 0,
  "udld_rfc5171_compatibility_mode": "string",
  "user_config": {}
}
```

5. Enabling the interface using all the attributes in the return body received from the GET request, modifying the `user_config` attribute to be: `"user_config":{"admin":"up"}`

```
$ curl -k --noproxy 192.0.2.5 -X PUT \
-b /tmp/auth_cookie -d '{
  "options": {},
  "other_config": {},
  "udld_arubaos_compatibility_mode": "forward_then_verify",
  "udld_compatibility": "aruba_os",
  "udld_enable": false,
  "udld_interval": 7000,
  "udld_retries": 4,
  "udld_rfc5171_compatibility_mode": "normal",
  "user_config": {
    "admin": "up"
  }
}' \
"https://192.0.2.5/rest/v1/system/interfaces/1%2F1%2F2"
```

In the preceding example, the following mutable attributes listed in the previous step were not included, so they are set to their default values, which could be empty:

- `description`
- `selftest_disable`

# Example: Changing an interface from layer 3 to layer 2

The following example is an excerpt of Python code that changes an existing interface from a layer 3 interface to a layer 2 interface.

To convert the interface, the Python code in this excerpt does the following:

1. Uses a GET request to get the configuration attributes of all the ports in the default VRF.

   In the configuration database, the set of ports are represented as a table.

2. Creates a JSON dictionary of the ports and their configuration attributes.
3. Removes the target port from the dictionary.

The `port` variable was defined using the standard port notation, which uses slashes. However, in REST URIs, the slash must be replaced by its percent-encoded equivalent, `%2F`. Therefore the variable used to represent the port in this code is `convert_port` instead of `port`.

4. Uses a PUT request to replace the port table in the configuration database with the modified information in the JSON dictionary.

   When this step completes, the layer 3 port 1/1/1 is removed from the configuration of the switch.

5. Defines the layer 2 port JSON data that will be used when creating the layer 2 port.

   When specifying the port in JSON data, you use the standard port notation using slashes, so the `port` variable is used instead of the `convert_port` variable.

6. Creates a layer 2 interface by using a POST request to add the target port to the `/system/ports` resource.

The code also handles logging into the switch, handling the session cookie, disabling switch certificate verification for every request (using the `verify=False` parameter), and logging out of the switch. The code in this excerpt uses the `requests` library.

```
import requests
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)


def main():
    ip_address = "192.168.2.101"
    credentials = {"username": "admin", "password": "admin123"}
    base_url = "https://{0}/rest/v1/".format(ip_address)
    # the target port
    port = "1/1/1"
    # port string must be converted
    convert_port = port.replace("/", "%2F")
    # requests session so cookie is persisted between calls
    s = requests.Session()
    vlan_mode = "access"
    vlan = 99
    try:
        # login
        s.post(base_url + "login", params=credentials, verify=False)
        # GET the existing layer 3 port table
        l3_ports = s.get(base_url + "system/vrfs/default?selector=configuration", verify=False)
        # create a dictionary from the l3 port table
        l3_dict = l3_ports.json()
        # remove the target port from the layer 3 port table dictionary
        l3_dict['ports'].remove("/rest/v1/system/ports/{}".format(convert_port))
        # PUT call to return the amended layer 3 port table to the switch
        s.put(base_url + "system/vrfs/default", json=l3_dict, verify=False)
        # layer 2 data for the target port
        l2port_data = {
            "referenced_by": "/rest/v1/system/bridge",
            "name": port,
            "vlan_mode": vlan_mode,
            "vlan_tag": "/rest/v1/system/bridge/vlans/" + str(vlan),
            "interfaces": ["/rest/v1/system/interfaces/{}".format(convert_port)],
            "admin": "up"
        }
        # POST new layer 2 target port
        new_int = s.post(base_url + "system/ports", json=l2port_data, verify=False)
    finally:
        # logout
        s.post(base_url + "logout", verify=False)
```

```
if __name__ == "__main__":
main()
```

# Examples: Interacting with a VSX peer switch

In the following examples, Virtual Switching Extension (VSX) is enabled, the primary VSX switch IP address is 192.0.2.5, and the secondary VSX switch IP address is 192.0.2.6:

- Getting the list of all VLANS on the connected switch at IP address 192.0.2.5:

  ```
  $ curl -k --noproxy 192.0.2.5 GET \
  -b /tmp/primary_auth_cookie \
  "https://192.0.2.5/rest/v1/system/bridge/vlans"
  ```

  Getting the list of all VLANs on the peer VSX switch:

  ```
  $ curl -k --noproxy 192.0.2.5 GET \
  -b /tmp/primary_auth_cookie \
  "https://192.0.2.5/vsx-peer/rest/v1/system/bridge/vlans"
  ```

- Getting the VSX status of the secondary VSX switch while connected to the primary VSX switch at IP address 192.0.2.5:

  ```
  $ curl -k --noproxy 192.0.2.5 GET \
  -b /tmp/primary_auth_cookie \
  "https://192.0.2.5/vsx-peer/rest/v1/system/vsx?attributes?oper_status
  ```

  You can also get the VSX status of the primary VSX switch while connected to the secondary VSX switch.

# https-server rest access-mode

### Syntax

```
https-server rest access-mode {read-only | read-write}
```

### Description

Changes the REST API access mode. The default mode is `read-only`. Changing the mode to `read-write` allows POST, PUT, and DELETE operations to be called on all configurable elements in the switch configuration database.

### Command context

```
config
```

### Parameters

**`read-only`**

Selects the read-only mode. Write access to most switch resources through the REST API is disabled. This value is the default value.

**`read-write`**

Selects the read/write mode. Allows POST, PUT, and DELETE methods to be called on all configurable elements in the switch database.

### Authority

Administrators

### Usage

Setting the mode to `read-write` on the REST API allows POST, PUT, and DELETE methods to be called on all configurable elements in the switch database.

The REST API in read/write mode is intended for use by advanced programmers who have a good understanding of the system schema and data relationships in the switch database.

**CAUTION:** Because the REST API in read/write mode can access every configurable element in the database, it is powerful but must be used with extreme caution: No semantic validation is performed on the data you write to the database, and configuration errors can destabilize the switch.

Some switch resources allow POST, PUT, and DELETE regardless of REST API mode.

The default is `read-only`.

### Example

```
switch(config)# https-server rest access-mode read-write
```

# https-server session close all

### Syntax

```
https-server session close all
```

### Description

Invalidates and closes all HTTPS sessions. All existing Web UI and REST sessions are logged out and all real-time notification feature WebSocket connections are closed.

### Command context

Manager (#)

### Authority

Administrators

### Usage

Typically, a user that has consumed the allowed concurrent HTTPS sessions and is unable to access the session cookie to log out manually must wait for the session idle timeout to start another session. This command is intended as a workaround to waiting for the idle timeout to close an HTTPS session. This command stops and starts the `hpe-restd` service, so using this command affects all existing REST sessions, Web UI sessions, and real-time notification subscriptions.

### Example

```
switch# https-server session close all
```

# https-server vrf

### Syntax

```
https-server vrf <VRF-NAME>

no https-server vrf <VRF-NAME>
```

### Description

Configures and starts the HTTPS server on the specified VRF. HTTPS server features include the REST API and the web user interfaces. By default, no VRFs have HTTPS servers configured or running.

The `no` form of the command stops any HTTPS servers running on the specified VRF and removes the HTTPS server configuration.

### Command context

```
config
```

### Parameters

**<VRF-NAME>**

Specifies the VRF name. Required. Length: Up to 32 alpha numeric characters.

### Authority

Administrators

### Usage

By using this command, you enable access to both the Web UI and to the REST API on the specified VRF. You can enable access on multiple VRFs.

By default, VRFs do not have HTTPS servers configured or running. Attempts to access web UI or REST URLs result in `404 Not Found` errors.

The VRF you select determines from which network the Web UI and REST API can be accessed.

For example:

- If you want to enable access to the REST API and Web UI through the OOBM port (management IP address), specify the built-in management VRF (`mgmt`).
- If you want to enable access to the REST API and Web UI through the data ports (for "inband management"), specify the built-in default VRF (`default`).
- If you want to enable access to the REST API and Web UI through only a subset of data ports on the switch, specify other VRFs you have created.

Aruba Network Analytics Engine scripts run in the default VRF, but you do not have to enable HTTPS server access on the default VRF for the scripts to run. If the switch has custom Aruba Network Analytics Engine scripts that require access to the Internet, for those scripts to perform those functions, you must configure a DNS name server must be configured on default VRF.

### Examples

- To enable access on all ports on the switch, specify the default VRF:

```
switch(config)# https-server vrf default
```

- To enable access on the OOBM port (management interface IP address), specify the management VRF:

```
switch(config)# https-server vrf mgmt
```

- To enable access on ports that are members of the VRF named `vrfprogs`, specify `vrfprogs`:

```
switch(config)# https-server vrf vrfprogs
```

- To enable access on the management port and ports that are members of the VRF named `vrfprogs`, enter two commands:

```
switch(config)# https-server vrf mgmt
switch(config)# https-server vrf vrfprogs
```

## show https-server

### Syntax

```
show https-server [vsx-peer]
```

### Description

Shows the status and configuration of the HTTPS server. The REST API and web user interface are accessible on VRFs that have the HTTPS server features configured only.

### Command context

Manager (#)

### Authority

Administrators

### Parameters

**[vsx-peer]**

Shows the output from the VSX peer switch. If the switches do not have the VSX configuration or the ISL is down, the output from the VSX peer switch is not displayed.

### Usage

Shows the configuration of the HTTPS server features.

**VRF**

Shows the VRFs, if any, for which HTTPS server features are configured.

**REST Access Mode**

Shows the configuration of the REST access mode:

**read-only**

Write access to most switch resources through the REST API is disabled. This value is the default value.

**read-write**

POST, PUT, and DELETE methods can be called on all configurable elements in the switch database.

### Examples

```
switch# show https-server

HTTPS Server Configuration
--------------------------

 VRF               : default, mgmt
 REST Access Mode  : read-only
```

| Response code | Description |
|---|---|
| 2*xx* | Successful operation |
| 200 | OK. Returned from GET and PUT operations, and nonconfiguration API calls such as Login or Logout. |
| 201 | Created. Returned from POST operations. |
| 204 | No Content. Returned from DELETE operations. |
| 4*xx* | Client-side error with error message returned |
| 400 | Bad request. Typically a problem with the request body, such as incorrectly formatted JSON, or the data violated a database constraint. |
| 401 | Unauthorized. No active session for this client (The login API has not been called), or too many sessions already created from this client. |
| 403 | Forbidden. Client session is valid but has no permissions to access the requested resource. |
| 404 | Not found. Resource does not exist, or the URI is incorrect for the desired resource. Can also occur if trying to access POST/PUT/DELETE API when the REST access-mode is set to read-only. |
| 5*xx* | Server-side error with error message returned |
| 500 | Internal server error. An unexpected error has occurred in processing the request. View the logs on the device for details. |
| 503 | Service unavailable. The device is receiving more requests than it can process and is defensively rejecting requests to protect resources. |

# General troubleshooting tips

- Connectivity is often the first issue you encounter. The REST API is enabled, or disabled, per VRF.

  To connect to the REST API through the management (OOBM) port, REST API access must be enabled on the management VRF. To connect to the REST API through a data port, REST API access must be enabled on the default VRF or a user-created VRF that includes that data port.

- Most resources do not allow POST, PUT, or DELETE methods and do not display those methods in the ArubaOS-CX REST API Reference unless the REST access mode is set to `read-write`.

- The JSON model of a resource can vary by method used. The JSON data you receive from the GET method is not the same as the JSON data you can or must provide with the POST or PUT methods:

  - The GET method model contains all the attributes.
  - The POST method model contains only the configuration attributes.
  - The PUT method model contains only the **mutable** (changeable) configuration attributes. If you do not provide all the mutable attributes in the request body of the PUT request, those attributes you do not provide are set to their defaults, which could be empty. If you attempt to provide an immutable attribute in a PUT request, an error is returned.

  You use the ArubaOS-CX REST API Reference to view information about the supported methods and resource models. You can also use the GET method with the `selector=configuration` parameter to get only the configuration attributes of a resource.

- Resources, attributes, and behaviors might differ between different versions of the REST API. Ensure that the version used in the URIs match the version running on the switch. The version is included in the URI prefix portion of the path of the URI. The REST API version running on the switch is shown at the bottom of the ArubaOS-CX REST API Reference browser window.

  - Example URI prefix: `https://192.0.2.5/rest/v1`
  - Example from the ArubaOS-CX REST API Reference browser window:

- Different switches have different hardware and features. For example, the management module resource ID is 1/1 for some switches, and 1/4 or 1/5 for other switches. To get information about the switch model, use the GET method request with the URI for the `platform_name` system attribute.

  For example:

  ```
  GET "https://192.0.2.5/rest/v1/system?attributes=platform_name"
  ```

  The following is an example of a response body for an Aruba 8320 switch:

  ```
  {
    "platform_name": "8320"
  }
  ```

  The following is an example of a response body for an Aruba 8400 switch:

```
{
  "platform_name": "8400X"
}
```

- You can additional platform-specific information through get requests for product information attributes or subsystem collections.

  Aruba 8400 switch examples:

  - Example request:

    ```
    GET "https://192.0.2.5/rest/v1/system/subsystems"
    ```

    Example response:

    ```
    [
      "/rest/v1/system/subsystems/fabric_card/1%2F3",
      "/rest/v1/system/subsystems/chassis/base",
      "/rest/v1/system/subsystems/line_card/1%2F3",
      "/rest/v1/system/subsystems/line_card/1%2F2",
      "/rest/v1/system/subsystems/line_card/1%2F8",
      "/rest/v1/system/subsystems/line_card/1%2F7",
      "/rest/v1/system/subsystems/line_card/1%2F10",
      "/rest/v1/system/subsystems/management_module/1%2F5",
      "/rest/v1/system/subsystems/rear_display_card/1%2FRDC",
      "/rest/v1/system/subsystems/fabric_card/1%2F2",
      "/rest/v1/system/subsystems/management_module/1%2F6",
      "/rest/v1/system/subsystems/line_card/1%2F9",
      "/rest/v1/system/subsystems/fan_tray/1%2F2",
      "/rest/v1/system/subsystems/fabric_card/1%2F1",
      "/rest/v1/system/subsystems/fan_tray/1%2F3",
      "/rest/v1/system/subsystems/line_card/1%2F1",
      "/rest/v1/system/subsystems/line_card/1%2F4",
      "/rest/v1/system/subsystems/fan_tray/1%2F1"
    ]
    ```

  - Example request:

    ```
    GET "https://192.0.2.5/rest/v1/system/subsystems/chassis/base?attributes=product_info"
    ```

    Example response:

    ```
    {
      "product_info": {
        "base_mac_address": "00:00:5E:00:53:00",
        "device_version": "",
        "instance": "1",
        "number_of_macs": "512",
        "part_number": "JL375A",
        "product_description": "8400 8-slot Chassis/3xFan Trays/18xFans/Cable Manager/X462 Bundle",
        "product_name": "8400 Base Chassis/3xFT/18xFans/Cbl Mgr/X462 Bundle",
        "serial_number": "SG00A2A00A",
        "vendor": "Aruba"
      }
    }
    ```

  Aruba 8320 switch examples:

  - Example request:

    ```
    GET "https://192.0.2.5/rest/v1/system/subsystems
    ```

    Example response:

    ```
    [
      "/rest/v1/system/subsystems/chassis/base",
      "/rest/v1/system/subsystems/line_card/1%2F1",
    ```

```
  "/rest/v1/system/subsystems/management_module/1%2F1"
]
```

- Example request:

  ```
  GET "https://10.102.254.250/rest/v1/system/subsystems/chassis/base?attributes=product_info"
  ```

  Example response:

  ```
  {
    "product_info": {
      "base_mac_address": "00:00:5E:00:53:01",
      "device_version": "",
      "instance": "1",
      "number_of_macs": "74",
      "part_number": "JL479A",
      "product_description": "8320",
      "product_name": "8320",
      "serial_number": "TW00000000",
      "vendor": "Aruba"
    }
  }
  ```

- The words "port" and "interface" have meanings that are different from other network operating systems. In the ArubaOS-CX operating system:
  - A port is the logical representation of a port.
  - An interface is the hardware representation of a port.
- You can enable debugging logs by using the `debug` command. The module name is `rest`. You can specify all severity log levels or a minimum severity log level.

  Example specifying all severity log levels:

  ```
  switch# debug rest all
  ```

  Example specifying a minimum severity log level of `error`:

  ```
  switch# debug rest all severity error
  ```

# Error: "'admin' password is not set"

### Symptom

An attempt to enable the HTTPS server using the `https-server vrf` command fails and the following error is returned:

```
Failed to enable https-server on VRF <VRF>. 'admin' password is not set
```

### Cause

The switch is shipped from the factory with a default user named `admin` without a password. The `admin` user must set a valid password before HTTPS servers can be enabled.

### Action

From the global configuration context, set a valid password for the `admin` user.

For example:

```
switch(config)# user admin password
Changing password for user admin
Enter password:************
Confirm password:************
```

# Error "certificate verify failed" returned from curl command

### Symptom

A curl command to the switch URL fails with an error similar to the following:

```
SSL3_GET_SERVER_CERTIFICATE:certificate verify failed
```

### Cause

The curl program could not verify the switch server certificate against the CA certificate bundle that comes with the curl installation, and you did not include the `-k` option in the curl command.

### Action

Retry the command with the `-k` option included.

The switch HTTPS server uses self-signed certificates, which cannot be verified against a certificate authority. The `-k` option disables curl certificate validation.

For example:

```
$ curl GET -k -b /tmp/auth_cookie \
"https://192.0.2.5/rest/v1/system/bridge/vlans"
```

# HTTP 400 error "Invalid Operation"

### Symptom

A REST request returns response code 400 and the response body contains the following text string:

```
Invalid operation
```

### Cause

The method used for this REST request is not supported for the specified resource. For example, the `Invalid operation` response is returned if you attempt a DELETE request on the `system` resource.

### Action

Use a method supported by the resource.

The ArubaOS-CX REST API Reference displays the methods supported by each resource.

# HTTP 400 error "Value is not configurable"

### Symptom

A PUT or POST request returns response code 400 and the response body contains the following text string:

`Value <value> is not configurable`

### Cause

The JSON data in the POST or PUT request body contains non-configuration or immutable attributes.

### Action

Retry the request with the correct JSON resource model for that PUT or POST method.

To determine which are the configuration attributes of a resource, you can send a GET request with the `selection=configuration` query parameter to the resource

You can also use the ArubaOS-CX REST API Reference to verify the JSON model of the PUT or POST method of the resource.

The category an attribute belongs to can depend on whether that instance of the resource is owned by the system or owned by a user. Configuration attributes can become status attributes in resource instances that are owned by the system. Status attributes can not be modified by users.

In addition, some configuration attributes cannot be changed after a resource is created. These immutable attributes cannot be included in a PUT request.

### More information

Categories of resource attributes on page 15

## HTTP 400 error "Reference failure"

### Symptom

A REST request returns response code 400 and the response body contains the following text string:

`Reference failure`

### Cause

You attempted to delete a resource that is referenced by other resources. Typically, this error occurs for resources that have no clear parent in the resource hierarchy, such as ports. For example, the `Reference failure` response is returned if you attempt a DELETE request on a port.

### Action

Remove all references to the resource.

After all references to a resource are removed, the resource is deleted automatically.

## HTTP 401 error "Authorization Required"

### Symptom

A REST request returns response code 401 and the response body contains the following text string:

`Authorization Required`

This response means that no valid session was found for the session token passed to the API.

### Solution 1

#### Cause

The user attempting the request is not logged into the REST API for one of the following reasons:

- The user has not yet logged in.
- The user logged in but the session has expired.

#### Action

Log in to the REST API.

### Solution 2

#### Cause

The user attempting the request is not logged in to the REST API because the user did not pass the correct session cookie to the API. Typically, incorrect session cookies are not a cause when accessing the REST API through a browser because the browser automatically handles the session cookie.

#### Action

1. Ensure that you save the session cookie returned from the login request.
2. Ensure that you pass the same cookie back to the switch with every REST API request, including the request to log out.

## HTTP 401 error "Login failed: session limit reached"

#### Symptom

A REST request or Web UI login attempt returns response code 401 and the response body contains the following text string:

```
Login failed: session limit reached
```

#### Cause

A user attempted to log into the REST API or the Web UI but that user already has the maximum number of concurrent sessions running.

#### Action

1. Log out from one of the existing sessions.

   Browsers share a single session cookie across multiple tabs or even windows. However, scripts that POST to the login resource without later posting to the logout resource can easily create the maximum number of concurrent sessions.

2. If the session cookie has been lost and it is not possible to log out of the session, wait for the session idle time limit to expire.

   When the session idle timeout expires, the session is terminated automatically.

3. If it is important enough to stop all HTTPS sessions on the switch instead of waiting for the session idle time limit to expire, you can stop all HTTPS sessions using the `https-server session close all` command.

This command stops and starts the `hpe-restd` service, so using this command affects all existing REST sessions, Web UI sessions, and real-time notification subscriptions.

**More information**

ArubaOS-CX REST API reference summary on page 17
https-server session close all on page 83

# HTTP 403 error "Forbidden" on a write request

**Symptom**

A POST, PUT, or DELETE REST request returns response code 403 and the response body contains the following text string:

```
Forbidden
```

**Cause**

The user attempting the request does not have administrator rights.

**Action**

Log in to the REST API with a user name that has administrator rights.

**More information**

User groups and access authorization on page 9

# HTTP 403 error "Forbidden" on a GET request

**Symptom**

A GET REST request returns response code 403 and the response body contains the following text string:

```
Forbidden
```

**Cause**

The user attempting the request is a member of the Auditors group, and the GET request specified a switch resource that users with auditor rights are not permitted to access.

**Action**

Log in to the REST API with a user name that has operator or administrator rights.

**More information**

User groups and access authorization on page 9

# HTTP 404 error when accessing the switch URL

**Symptom**

The switch is operational and you are using the correct URL for the switch, but attempts to access the REST API or Web UI result in an HTTP 404 "Page not found" error.

### Cause

REST API access is not enabled on the VRF that corresponds to the access port you are using. For example, you are attempting to access the REST API or Web UI from the management (OOBM) port, and access is not enabled on the `mgmt` VRF.

### Action

Use the `https-server vrf` command to enable REST API access on the specified VRF.

For example:

```
switch(config)# https-server vrf mgmt
```

### More information

Enabling access to the REST API on page 19

## HTTP 404 error "Object not found" when using a write method

### Symptom

A PUT or DELETE request returns response code 404 and the response body contains the following text string:

```
Object not found
```

### Cause

The resource does not exist in the system. The URI in the request is incorrect or the resource has not been added to the configuration.

### Action

Verify the URI of the resource you are attempting to change or delete and retry the request.

## HTTP 404 error "Page not found" when using a write method

### Symptom

Using the GET method is successful, but attempting a POST, PUT, or DELETE method results in an HTTP 404 "Page not found" error.

### Cause

The REST API access mode is set to `read-only`.

### Action

Set the REST API access mode to `read-write`.

```
switch(config)# https-server rest access-mode read-write
```

Enabling the read/write mode on the REST API allows POST, PUT, and DELETE operations to be called on all configurable elements in the switch database.

### More information

Setting the REST API access mode to read/write on page 19

# Logout fails

### Symptom

An attempt to log out of the REST API from a script or curl command fails.

### Cause

The session cookie was not supplied or does not contain the correct session token.

### Action

- Repeat the command and send the correct session cookie or modify the script to send the correct session cookie.
- If the session cookie has been lost and it is not possible to log out of the session, wait for the session idle time limit to expire.

  When the session idle timeout expires, the session is terminated automatically.

**Networking Websites**

**Hewlett Packard Enterprise Networking Information Library**

**www.hpe.com/networking/resourcefinder**

**Hewlett Packard Enterprise Networking Software**

**www.hpe.com/networking/software**

**Hewlett Packard Enterprise Networking website**

**www.hpe.com/info/networking**

**Hewlett Packard Enterprise My Networking website**

**www.hpe.com/networking/support**

**Hewlett Packard Enterprise My Networking Portal**

**www.hpe.com/networking/mynetworking**

**Hewlett Packard Enterprise Networking Warranty**

**www.hpe.com/networking/warranty**

**General websites**

**Hewlett Packard Enterprise Information Library**

**www.hpe.com/info/EIL**

For additional websites, see **Support and other resources**.

# Accessing Hewlett Packard Enterprise Support

- For live assistance, go to the Contact Hewlett Packard Enterprise Worldwide website:

  **http://www.hpe.com/assistance**
- To access documentation and support services, go to the Hewlett Packard Enterprise Support Center website:

  **http://www.hpe.com/support/hpesc**

**Information to collect**

- Technical support registration number (if applicable)
- Product name, model or version, and serial number
- Operating system name and version
- Firmware version
- Error messages
- Product-specific reports and logs
- Add-on products or components
- Third-party products or components

# Accessing updates

- Some software products provide a mechanism for accessing software updates through the product interface. Review your product documentation to identify the recommended software update method.
- To download product updates:

  **Hewlett Packard Enterprise Support Center**

  **www.hpe.com/support/hpesc**

  **Hewlett Packard Enterprise Support Center: Software downloads**

  **www.hpe.com/support/downloads**

  **Software Depot**

  **www.hpe.com/support/softwaredepot**
- To subscribe to eNewsletters and alerts:

  **www.hpe.com/support/e-updates**
- To view and update your entitlements, and to link your contracts and warranties with your profile, go to the Hewlett Packard Enterprise Support Center **More Information on Access to Support Materials** page:

  **www.hpe.com/support/AccessToSupportMaterials**

**IMPORTANT:** Access to some updates might require product entitlement when accessed through the Hewlett Packard Enterprise Support Center. You must have an HPE Passport set up with relevant entitlements.

# Customer self repair

Hewlett Packard Enterprise customer self repair (CSR) programs allow you to repair your product. If a CSR part needs to be replaced, it will be shipped directly to you so that you can install it at your convenience. Some parts do not qualify for CSR. Your Hewlett Packard Enterprise authorized service provider will determine whether a repair can be accomplished by CSR.

For more information about CSR, contact your local service provider or go to the CSR website:

**http://www.hpe.com/support/selfrepair**

# Remote support

Remote support is available with supported devices as part of your warranty or contractual support agreement. It provides intelligent event diagnosis, and automatic, secure submission of hardware event notifications to Hewlett Packard Enterprise, which will initiate a fast and accurate resolution based on your product's service level. Hewlett Packard Enterprise strongly recommends that you register your device for remote support.

If your product includes additional remote support details, use search to locate that information.

**Remote support and Proactive Care information**

**HPE Get Connected**

**www.hpe.com/services/getconnected**

**HPE Proactive Care services**

**www.hpe.com/services/proactivecare**

**HPE Proactive Care service: Supported products list**

**www.hpe.com/services/proactivecaresupportedproducts**

**HPE Proactive Care advanced service: Supported products list**

**www.hpe.com/services/proactivecareadvancedsupportedproducts**

**Proactive Care customer information**

**Proactive Care central**

**www.hpe.com/services/proactivecarecentral**

**Proactive Care service activation**

**www.hpe.com/services/proactivecarecentralgetstarted**

# Warranty information

To view the warranty information for your product, see the links provided below:

**HPE ProLiant and IA-32 Servers and Options**

**www.hpe.com/support/ProLiantServers-Warranties**

**HPE Enterprise and Cloudline Servers**

**www.hpe.com/support/EnterpriseServers-Warranties**

**HPE Storage Products**

**www.hpe.com/support/Storage-Warranties**

**HPE Networking Products**

**www.hpe.com/support/Networking-Warranties**

# Regulatory information

To view the regulatory information for your product, view the *Safety and Compliance Information for Server, Storage, Power, Networking, and Rack Products*, available at the Hewlett Packard Enterprise Support Center:

**www.hpe.com/support/Safety-Compliance-EnterpriseProducts**

### Additional regulatory information

Hewlett Packard Enterprise is committed to providing our customers with information about the chemical substances in our products as needed to comply with legal requirements such as REACH (Regulation EC No 1907/2006 of the European Parliament and the Council). A chemical information report for this product can be found at:

**www.hpe.com/info/reach**

For Hewlett Packard Enterprise product environmental and safety information and compliance data, including RoHS and REACH, see:

**www.hpe.com/info/ecodata**

For Hewlett Packard Enterprise environmental information, including company programs, product recycling, and energy efficiency, see:

**www.hpe.com/info/environment**

# Documentation feedback

Hewlett Packard Enterprise is committed to providing documentation that meets your needs. To help us improve the documentation, send any errors, suggestions, or comments to Documentation Feedback (**docsfeedback@hpe.com**). When submitting your feedback, include the document title, part number, edition, and publication date located on the front cover of the document. For online help content, include the product name, product version, help edition, and publication date located on the legal notices page.