Troubleshoot API-based EPNM Notifications

Contents

Introduction

This document describes how to troubleshoot EPNM Notifications when REST API is used to access device fault information.

Background Information

The client you implement must be capable of handling and subscribing to any of the two mechanisms used by the Evolved Programmable Network Manager (EPNM) to send notifications.

EPNM API Notifications

Notifications alert network administrators and operators about important events or issues related to the network. These notifications help ensure that potential problems are detected and resolved quickly, which reduces downtime and improves overall network performance.

EPNM can handle different methods, such as notifications via e-mail, Simple Network Management Protocol (SNMP) traps to specified receivers, or Syslog messages to external Syslog servers. In addition to these methods, EPNM also provides a Representational State Transfer Application Programming Interface (REST API) that can be used in order to retrieve information about inventory, alarms, service activation, template execution, and High-Availability.

API-based notifications are currently supported with the use of two different mechanisms:

- Connection-oriented notifications: The client subscribes to a predefined URL and uses a WebSocket client with basic authentication through a secure HTTPS channel.
- Connectionless notifications: The user is expected to have a REST web service that is capable of accepting extensible markup language (XML) and/or JavaScript Object Notation (JSON) payloads as a POST request.

All the notifications share the same schema and can be retrieved in JSON or XML formats.

Basic EPNM Configuration

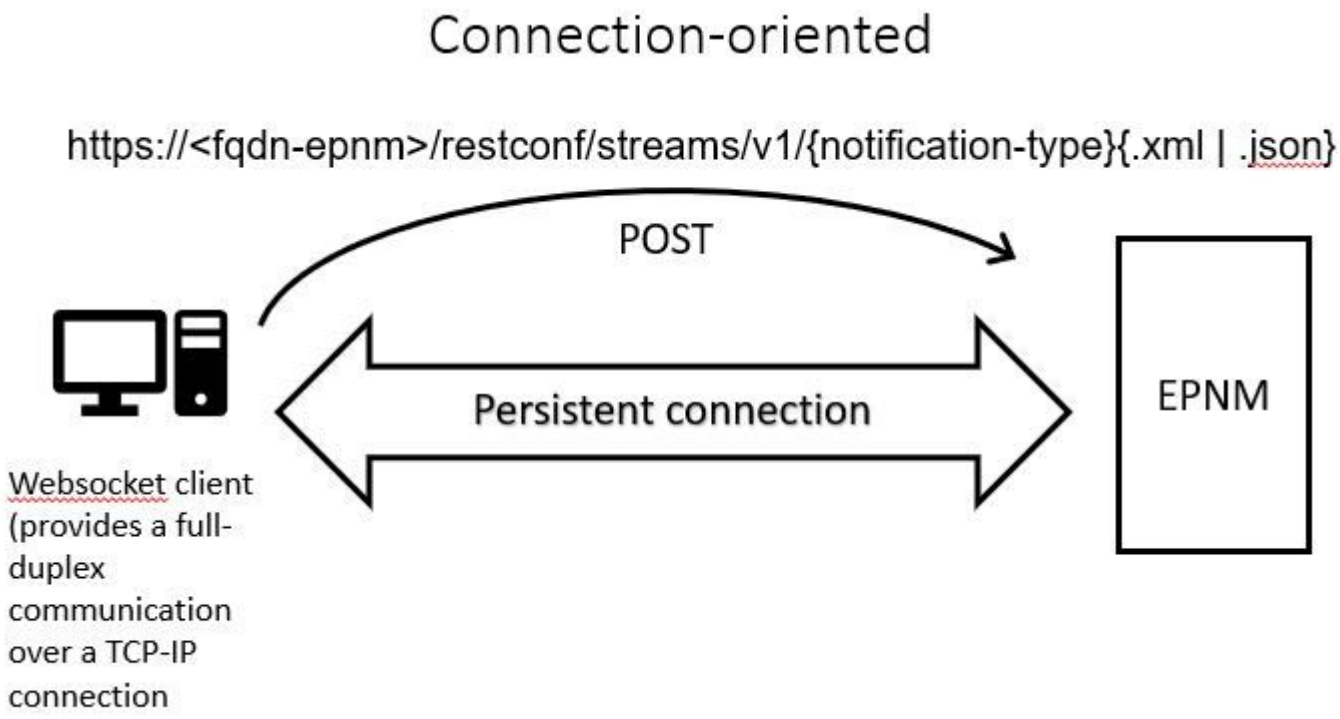
By default, alarm and inventory notifications are disabled. In order to enable them, change the restconf-config.properties file as indicated (it is not necessary to restart the EPNM application):

```
/opt/CSCOlumos/conf/restconf/restconf-config.properties

epnm.restconf.inventory.notifications.enabled=true
epnm.restconf.alarm.notifications.enabled=true
```

Connection-Oriented Notifications

In the picture, the client machine runs a WebSocket and subscribes to the EPNM with a predefined URL, with basic authentication, and through a secure HTTPS channel.



Run a WebSockets Python Client

The WebSocket-client library in Python can be used to create a WebSocket in the client machine.

```
import websocket
import time
import ssl
import base64

def on_message(ws, message):
    print(message)
```

```
def on_error(ws, error):
    print(error)

def on_close(ws, close_status_code, close_msg):
    print("### closed \###")

def on_open(ws):
    ws.send("Hello, Server!")

if __name__ == "__main__":
    username = "username"
    password = "password"
    credentials = base64.b64encode(f"{username}:{password}".encode("utf-8")).decode("utf-8")
    headers = {"Authorization": f"Basic {credentials}"}
    websocket.enableTrace(True)
    ws = websocket.WebSocketApp("wss://10.122.28.3/restconf/streams/v1/inventory.json",
                                on_message=on_message,
                                on_error=on_error,
                                on_close=on_close,
                                header=headers)
    ws.on_open = on_open
    ws.run_forever(sslopt={"cert_reqs": ssl.CERT_NONE})
```

Subscription of a Connection-Oriented Client

This code sets up a WebSocket client that subscribes to EPNM at **wss://10.122.28.3/restconf/streams/v1/inventory.json**. It uses the Python WebSocketlibrary in order to establish the connection and handle in and out messages. The subscription can also be (on the basis of what kind of notification you want to subscribe to):

- /restconf/streams/v1/alarm{.xml | .json}
- /restconf/streams/v1/service-activation{.xml | .json}
- /restconf/streams/v1/template-execution{.xml | .json}
- /restconf/streams/v1/all{.xml | .json}

The on_message, on_error and on_close functions are callback functions that are called when the WebSocket connection receives a message, encounters an error, or is closed, respectively. The **on_open** function is a callback that is called when the WebSocket connection is established and ready to use.

The username and password variables are set to the login credentials required to access the remote server. These credentials are then encoded with the base64 module and added to the headers of the WebSocket request.

The run_forever method is called on the WebSocket object in order to start the connection, keep it open indefinitely, and listen for messages that come from the server. The sslopt parameter is used to configure the SSL/TLS options for the connection. The CERT_NONE flag disables certification validation.

Run the code In order to have the WebSocket ready to receive the notifications:

```
(env) devasc@labvm:~/epnm$ python conn-oriented.py
--- request header ---
GET /restconf/streams/v1/inventory.json HTTP/1.1
Upgrade: websocket
Host: 10.122.28.3
Origin: https://10.122.28.3
Sec-WebSocket-Key: YYYYYYYYYYY
```

```
Sec-WebSocket-Version: 13
Connection: Upgrade
Authorization: Basic XXXXXXXXXXXX


-----------------------
--- response header ---
HTTP/1.1 101
Set-Cookie: JSESSIONID=5BFB68B0126226A0A13ABE595DC63AC9; Path=/restconf; Secure; HttpOnly
Strict-Transport-Security: max-age=31536000;includeSubDomains
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Upgrade: websocket
Connection: upgrade
Sec-WebSocket-Accept: Ozns7PGgHjrXj0nAgnlhbyVKPjc=
Date: Thu, 30 Mar 2023 16:18:19 GMT
Server: Prime
-----------------------
Websocket connected
++Sent raw: b'\x81\x8es\x99ry;\xfc\x1e\x15\x1c\xb5R*\x16\xeb\x04\x1c\x01\xb8'
++Sent decoded: fin=1 opcode=1 data=b'Hello, Server!'
++Rcv raw: b'\x81\x0eHello, Server!'
++Rcv decoded: fin=1 opcode=1 data=b'Hello, Server!'
Hello, Server!
```

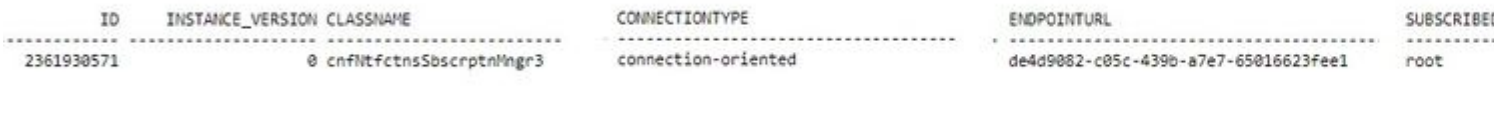
You can check the notification subscriptions to the server with this DB query:

```
ade # ./sql_execution.sh "SELECT * from RstcnfNtfctnsSbscrptnMngr WHERE CONNECTIONTYPE = 'connection-ori
```

In order to better visualize the conn-oriented.txt file (which is the result of the DB query), you can convert it to HTML using a tool like aha (here its use is illustrated in an Ubuntu machine):

```
devasc@labvm:~/tmp$ sudo apt-get install aha
devasc@labvm:~/tmp$ cat conn-oriented.txt | aha > conn-oriented.html
```

Then open the conn-oriented.html file in a browser:



ID	INSTANCE_VERSION	CLASSNAME	CONNECTIONTYPE	ENDPOINTURL	SUBSCRIBE
2361930571	0	cnfNtfctnsSbscrptnMngr3	connection-oriented	de4d9082-c05c-439b-a7e7-65016623fee1	root

From the EPNM online documentation, once established, the same connection is kept alive throughout the lifecycle of the application:

- until the client disconnects from the server
- until the server goes down either for maintenance or during a failover

If, for some reason, you need to delete a specific subscription, you can send an HTTP DELETE request with the SUBSCRIPTIONID specified in the URL https://<fqdn-epnm/restconf/data/v1/cisco-

notifications:subscription/{SUBSCRIPTIONID}. For example:

```
devasc@labvm:~/tmp$ curl --location --insecure --request DELETE 'https://10.122.28.3/restconf/data/v1/ci
> --header 'Accept: application/json' \
> --header 'Content-Type: application/json' \
> --header 'Authorization: Basic XXXXXXXX'
```

Verification of Messages, DEBUG Entries, show log**, Filename Used, SQL Outputs**

In order to troubleshoot why a client that uses a connection-oriented mechanism does not properly receive notifications, you can run the indicated DB query and check if the subscription is present or not. If it is not present, ask the client owner to ensure to issue the subscription.

In the meantime, you can enable the DEBUG level in com.cisco.nms.nbi.epnm.restconf.notifications.handler.NotificationsHandlerAdapter so you can catch it whenever the subscription is sent:

```
ade # sudo /opt/CSCOlumos/bin/setLogLevel.sh com.cisco.nms.nbi.epnm.restconf.notifications.handler.Notif
Loglevel set to DEBUG for com.cisco.nms.nbi.epnm.restconf.notifications.handler.NotificationsHandlerAdap
```

After the subscription is sent, you can check if an entry with the IP address of the WebSocket client appears in localhost_access_log.txt:

```
ade # zgrep -h '"GET /restconf/streams/.* HTTP/1.1" 101' $(ls -1t /opt/CSCOlumos/logs/localhost_access_l
10.82.244.205 - - [28/Aug/2023:16:13:03 -0300] "GET /restconf/streams/v1/inventory.json HTTP/1.1" 101 -
10.82.244.205 - - [28/Aug/2023:22:17:05 -0300] "GET /restconf/streams/v1/inventory.json HTTP/1.1" 101 -
```

Finally, check again the DB (notice that the timestamp matches the entry in localhost_access_log.txt).

H	CONNECTIONTYPE	ENDPOINTURL	ISENDPOINTREACHABLE	NOTIFICATIONFORMAT	SUBSCR
	connection-oriented	852a674a-e3d0-4ecc-8ea0-787af30f1305	0	json	root

The next log shows when the POST requests for subscriptions are sent:

```
ade # grep -Eh 'DEBUG com.cisco.nms.nbi.epnm.restconf.notifications.handler.NotificationsHandlerAdapter
2023-08-28 22:17:06,221: DEBUG com.cisco.nms.nbi.epnm.restconf.notifications.handler.NotificationsHandle
2023-08-28 22:17:06,221: DEBUG com.cisco.nms.nbi.epnm.restconf.notifications.handler.NotificationsHandle
2023-08-28 22:17:06,221: DEBUG com.cisco.nms.nbi.epnm.restconf.notifications.handler.NotificationsHandle
2023-08-28 22:17:06,221: DEBUG com.cisco.nms.nbi.epnm.restconf.notifications.handler.NotificationsHandle
```

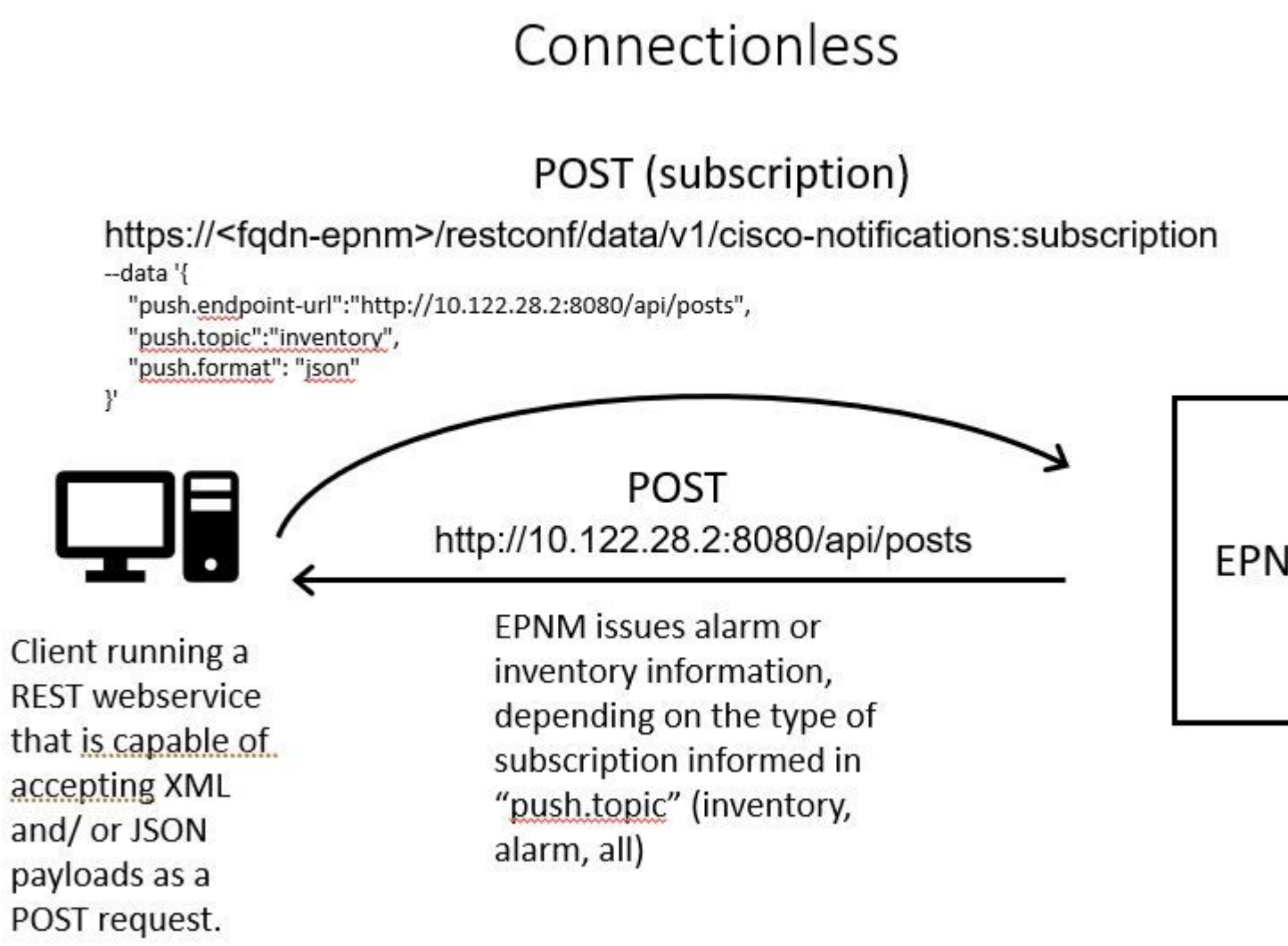
As long as the connection is kept alive, a notification of type push-change-update is sent from the EPN-M

server to all clients that subscribed for notifications. The example shows one of the notifications that are sent by the EPNM when the hostname of an NCS2k is changed:


```
{
  "push.push-change-update":{
    "push.notification-id":2052931975556780123,
    "push.topic":"inventory",
    "push.time-of-update":"2023-03-31 13:50:36.608",
    "push.time-of-update-iso8601":"2023-03-31T13:50:39.681-03:00",
    "push.operation":"push:modify",
    "push.update-data":{
      "nd.node":{
        "nd.description":"SOFTWARE=ONS,IPADDR=10.10.1.222,IPMASK=255.255.255.0,DEFRTR=255.255.255.255,IP
        "nd.equipment-list":"",
        "nd.fdn":"MD=CISCO_EPNM!ND=tcc222c",
        "nd.sys-up-time":"217 days, 14:40:170.00"
      }
    }
  }
}
```


Connectionless Notifications

The next is the workflow in the case of connectionless notifications:



Run a REST Webservice Python Client

The user is expected to have a REST web service that is capable of accepting XML and/or JSON payloads as a POST request. This REST service is the endpoint to which the Cisco EPNMrestconf notifications framework publishes notifications. This is an example of a REST web service to be installed in the remote machine:

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@ app.route('/api/posts', methods=['POST'])
def create_post():
    post_data = request.get_json()
    response = {'message': 'Post created successfully'}
    print(post_data)
    return jsonify(response), 201

if __name__ == '__main__':
    app.run(debug=True, host='10.122.28.2', port=8080)
```

This is a Python Flask web application that defines a single endpoint /api/posts that accepts **HTTP POST** requests. The create_post() function is called whenever an **HTTP POST** request is made to /api/posts. Inside the create_post() function, the data from the request that comes in is retrieved with the use of **request.get_json(),** which returns a dictionary of the JSON payload. The payload is then printed with **print(post_data)** for debug purposes. After that, a response message is created with the key message and value **Post created successfully** (in dictionary format). This response message is then returned to the client with an HTTP status code of 201 (created).

The if __name__ == '__main__': block is a standard Python construct that checks if the script runs as the main program, rather than imported as a module. If the script runs as the main program, it starts the Flask application and runs it on the specified IP address and port. The **debug=True** argument enables debug mode, which provides detailed error messages and automatic reloading of the server when changes are made to the code.

Run the program to start the REST web service:

```
(venv) [apinelli@centos8_cxlabs_spo app]$ python connectionless.py
 * Serving Flask app 'connectionless' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://10.122.28.2:8080/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 117-025-064
```

Subscription of a Connection-less Client

The user subscribes to notifications: the RESTservice endpoint is sent along with the topic in order to subscribe to. In this case, the topic is all.

```
[apinelli@centos8_cxlabs_spo ~]$ curl --location -X POST --insecure 'https://10.122.28.3/restconf/data/v
> --header 'Accept: application/json' \
> --header 'Content-Type: application-json' \
> --header 'Authorization: Basic XXXXXXXXX' \
> --data '{
>     "push.endpoint-url":"http://10.122.28.2:8080/api/posts",
>     "push.topic":"all",
>     "push.format": "json"
> }'
```

The expected response is a 201 response, along with the details from the subscription in the body of the response:

```
{
  "push.notification-subscription": {
    "push.subscription-id": 7969974728822328535,
    "push.subscribed-user": "root",
```

```
    "push.endpoint-url": "http:\/\/10.122.28.2:8080\/api\/posts",
    "push.topic": "all",
    "push.creation-time": "Tue Aug 29 10:02:05 BRT 2023",
    "push.creation-time-iso8601": "2023-08-29T10:02:05.887-03:00",
    "push.time-of-update": "Tue Aug 29 10:02:05 BRT 2023",
    "push.time-of-update-iso8601": "2023-08-29T10:02:05.887-03:00",
    "push.format": "json",
    "push.connection-type": "connection-less"
  }
}
```

It is possible to get the list of notifications the user is subscribed to with a GET request:

```
curl --location --insecure 'https://10.122.28.3/restconf/data/v1/cisco-notifications:subscription' \
--header 'Accept: application/json' \
--header 'Content-Type: application/json' \
--header 'Authorization: Basic XXXXXXXXXXX'
```

The response yielded is as:

```
{
    "com.response-message": {
        "com.header": {
            "com.firstIndex": 0,
            "com.lastIndex": 1
        },
        "com.data": {
            "push.notification-subscription": [
                {
                    "push.subscription-id": 2985507860170167151,
                    "push.subscribed-user": "root",
                    "push.endpoint-url": "http://10.122.28.2:8080/api/posts",
                    "push.session-id": 337897630,
                    "push.topic": "inventory",
                    "push.creation-time": "Fri Mar 31 17:45:47 BRT 2023",
                    "push.time-of-update": "Fri Mar 31 17:45:47 BRT 2023",
                    "push.format": "json",
                    "push.connection-type": "connection-less"
                },
                {
                    "push.subscription-id": 7969974728822328535,
                    "push.subscribed-user": "root",
                    "push.endpoint-url": "http://10.122.28.2:8080/api/posts",
                    "push.session-id": 0,
                    "push.topic": "all",
                    "push.creation-time": "Tue Aug 29 10:02:05 BRT 2023",
                    "push.time-of-update": "Tue Aug 29 10:02:05 BRT 2023",
                    "push.format": "json",
                    "push.connection-type": "connection-less"
                }
            ]
        }
    }
}
```

Verification of Messages, DEBUG Entries, show log, **Filename used, SQL Outputs**

Notice from the response that there are two subscriptions: one for all (**"push.topic": "all"**) and one for inventory (**"push.topic": "inventory"**). You can confirm it with a query to the database (notice that the type of subscription is 'connection-less' and the SUBSCRIPTIONID fields match the output of the GET command as highlighted in yellow):

```
ade # ./sql_execution.sh "SELECT * from RstcnfNtfctnsSbscrptnMngr WHERE CONNECTIONTYPE = 'connection-les
```

```
        ID     INSTANCE_VERSION CLASSNAME                CONNECTIONTYPE     ENDPOINTURL                       SUBSCRIBEDUSER  SUBSCRIPTIONCREATIONTIME
---------- -------------------- ------------------------ ------------------ --------------------------------- --------------- ------------------------
2361930573                    0 cnfNtfctnsSbscrptnMngr3  connection-less    http://10.122.28.2:8080/api/posts root            Tue Aug 29 10:02:05 BRT 2
 337897630                    0 cnfNtfctnsSbscrptnMngr3  connection-less    http://10.122.28.2:8080/api/posts root            Fri Mar 31 17:45:47 BRT 2
```

If you need to delete a connectionless subscription, you can send an HTTP DELETE request, with the subscription ID you want to delete. Suppose you want to delete **subscription-id 2985507860170167151:**

```
curl --location --insecure --request DELETE 'https://10.122.28.3/restconf/data/v1/cisco-notifications:su
--header 'Accept: application/json' \
--header 'Content-Type: application-json' \
--header 'Authorization: Basic XXXXXXXXXX'
```

Now if you query the DB again, you see only the subscription with SUBSCRIPTIONID equal to 7969974728822328535.

When a change in inventory occurs, the client prints the notifications (which are of the same type as the connection-oriented notifications seen in the section about connected-oriented clients), followed by the 201 response:

```
(venv) [apinelli@centos8_cxlabs_spo app]$ python connectionless.py
 * Serving Flask app 'connectionless' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://10.122.28.2:8080/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 117-025-064
{'push.push-change-update': {'push.notification-id': -2185938612268228828, 'push.topic': 'inventory', 'p
10.122.28.3 - - [31/Mar/2023 16:47:23] "POST /api/posts HTTP/1.1" 201 -
{'push.push-change-update': {'push.notification-id': -1634959052215805274, 'push.topic': 'inventory', 'p
10.122.28.3 - - [31/Mar/2023 16:47:27] "POST /api/posts HTTP/1.1" 201 -
```

Conclusion

In this document, the two types of API-based notifications that are possible to be configured in EPNM (connectionlessand connection-oriented) are explained and the examples of the respective clients that can be used as a base for simulation purposes are given.

Related Information

- https://www.cisco.com/c/dam/en/us/td/docs/net_mgmt/epn_manager/RESTConf/Cisco_Evolved_Programmab
- Technical Support & Documentation - Cisco Systems