IBM Spectrum Virtualize scripting

Mikhail Zakharov mikhail.zakharov@cz.ibm.com

Revision history

1.0	2017.11.01	Initial version
1.1	2019.05.02	HTTPS on IBM Spectrum Virtualize updated

Preface

This topic touches some universal ways of accessing IBM Spectrum Virtualize system with scripts. These methods can be easily applied to reporting, configuration and storage administration tasks. Though most of the current operating systems and almost any modern programming language have resources to illustrate the subject, this section concentrates mostly on UNIX/Linux environments with Bourne-again shell (bash) for simple examples and on Python version 3 for advanced cases.

This section demonstrates basic usage of the following protocols, standards and APIs:

- Secure Shell (SSH)
- SMI-S
- HTTPS and RESTful API on IBM Spectrum Control
- HTTPS on IBM Spectrum Virtualize

Secure Shell (SSH)

Secure Shell (SSH) is a network protocol for secure operations on remote services over insecure network.

IBM Spectrum Virtualize has powerful command-line interface (CLI) which is accessible via SSH protocol. Together with SSH, Spectrum Virtualize CLI can be used for interactive system configuration, storage administration and reporting as well as for automating these tasks with scripts.

Attention: SSH protocol allows to authenticate users by passwords or by asymmetric cryptography methods – SSH-keys. It is strongly advised to configure and use SSH-keys for security reasons.

Bash

Typically, under UNIX-like environments bash scripts are written for default SSH client (**ssh**) installed in the system.

Most often, SSH client is used in the form of executing a specified command, but not a login shell on a remote system. In simple words, instead of opening an interactive session, **ssh** executes a command on the remote system, forwards its output to the local computer and finally exits. This mode allows to use **ssh** in shell command substitutions and/or in conveyors for piping its standard output to any external program for further filtering, parsing and data processing.

The first line of Example 1 shows how to use ssh to execute a command (**lssystemstats**) on the remote IBM Spectrum Virtualize system (**itso_storage**) with privileges of **itso_user** and store its output in a local shell variable (**lsstats**) by using shell command substitution mechanism.

The second line of the example demonstrates further data processing: **printf** command pipes contents of the **sys_data** variable through the **grep** filter to print out only vdisk related lines:

```
lsstats=$(ssh itso_user@itso_storage lssystemstats)
printf "$lsstats" | grep vdisk
```

Example 1 Using ssh to extract data from the storage system and save it in a variable

If there is no need to store data for the further processing, the previous code can be simplified using pipes only:

```
ssh itso_user@itso_storage lssystemstats | grep vdisk
```

Example 2 Extracting data from the storage system to process it with a conveyor of commands

For simple cases on Windows, plink.exe – SSH command line client from PuTTY family utilities combined with **findstr** command can be used to achieve similar results:

```
C:\Program Files\PuTTY>plink itso_user@itso_storage lssystemstats | findstr vdisk
```

Example 3 **plink** usage

Note, another good reason to use SSH-key based authentication in scripts is automatic key submission. SSH client will take care of it during authentication process on the remote system. Otherwise, with password based methods, a user will have to enter passwords manually on every ssh connection. Though, if key-based authentication is not available in a particular environment, it is still possible to modify the bash script to wrap ssh command by a special utility that supplies user password automatically, this method can be potentially insecure and it is out of the scope of this chapter.

In general, the method of execution commands remotely via ssh opens a green way to write advanced scripts for fetching data from IBM Spectrum Virtualize systems and processing it on a local computer with tools and utilities available to a user in a particular environment.

Example 4 shows a short script which endlessly (with an interval of 1 second) collects current system-level performance statistic and saves the acquired data into a nice CSV-formatted file:

```
#!/bin/bash

user="itso_user"
target="itso_storage"
fs=","
cmd="lssystemstats -nohdr -delim "${fs}
header="Date"${fs}"Name"${fs}"Current value"
outfile="sysstats.csv"
frequency=1

printf '%s\n' "$header" > $outfile
while true; do
   ssh ${user}@${target} ${cmd} |
      gawk -F${fs} 'BEGIN {OFS=FS} {print strftime("%F %T"), $1, $2}' >> $outfile
   sleep $frequency
```

```
done
```

Example 4   Collecting system-level performance statistic in a file

In the first part of the script various variables are defined to be used lately:

- **user='itso_user'** – specifies a username on a storage system
- **target='itso_storage'** – the IBM Spectrum Virtualize storage system itself
- **fs=','** – is a variable to hold a field separator, and comma sign is a good choice for formatted textual data
- **cmd="lssystemstats -nohdr -delim "${fs}** – the command to be executed on the storage system
- **header="Name"${fs}"Current value"** – custom header to be printed instead of default one
- **outfile='sysstats.txt'** – the file name for output data
- **frequency=1** – a statistic collection interval in seconds

Also note, most of IBM Spectrum Virtualize CLI commands have these handy options:

- **-delim** to specify a custom field separator
- **-nohdr** to suppress headers in the output.

The second part of the script is an infinite **while** loop which collects and processes data. In Example 4, GNU implementation of **awk** language (**gawk**) is used for this purpose as a tool to parse data by columns and add current timestamp into each row.

The main disadvantage of Example 4 script is that a new SSH connection with the storage system is established on every loop iteration. It becomes really painful for password-based SSH authentication as the user has to type a valid password continuously, but even with key-based authentication this is not optimal as it takes additional time and system resources to reopen connections.

Fortunately, IBM Spectrum Virtualize itself is powerful enough to execute sophisticated scripts directly in the shell of the storage system (actually this shell is bash, running in restricted mode – rbash).

A slightly improved version of system-level performance collecting script of Example 5 shows the way to offload ssh loop from a local computer to the shell on the remote IBM Spectrum Virtualize system while post-processing performance data with **gawk** stays on the local computer:

```
#!/bin/bash

user="itso_user"
target="itso_storage"
fs=","
cmd="lssystemstats -nohdr -delim "${fs}
header="Date"${fs}"Name"${fs}"Current value"
outfile="sysstats.csv"
frequency=1

printf '%s\n' "$header" > $outfile
ssh ${user}@${target} "while true; do $cmd; sleep $frequency; done" |
   gawk -F${fs} 'BEGIN {OFS=FS} {print strftime("%F %T"), $1, $2}' >> $outfile
```

Example 5   Moving the performance collection loop to the storage system

The advantage of this method is that there is a single connection only established with IBM Spectrum Virtualize as the while loop runs in the shell of the remote storage system.

Python

In Python an external module is required to connect with IBM Spectrum Virtualize via SSH protocol. Nowadays, the most popular one is open-source **paramiko** and it can be installed using **pip**.

Example 6 illustrates the simplest template of **paramiko** usage in a Python script:

```
import paramiko

target = 'itso_storage'
user = 'itso_user'
command = 'lsvdisk'

client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect(hostname=target, username=user)

stdin, stdout, stderr = client.exec_command(command)
data = stdout.read()
errors = stderr.read()

if data:
    print(data.decode('US-ASCII'))

if errors:
    print(errors.decode('US-ASCII'))
```

Example 6   Basic **paramiko** usage

Here, in the first line of the script **paramiko** module is loaded and three next lines define variables:

- **target** – a hostname/IP address of IBM Spectrum Virtualize system. In this script it is “itso_storage”.
- **user** – a username (itso_user) to log into the storage system
- **command** – a command (lsvdisk) to run on the target.

Further, paramiko.SSHClient() represents a session with SSH server.

The **set_missing_host_key_policy(paramiko.AutoAddPolicy())** defines what to do if the remote system fingerprints are not known locally. Actually there are two policies: **AutoAddPolicy** and **RejectPolicy**. We chose first one to simplify the example and to connect with the remote storage system anyway, but for security reasons in production and in advanced scripts, **load_system_host_keys()** can be used to load system host keys.

Next, **connect(hostname=target, username=user)** connects with the storage system. If keys-based authentication is configured properly, keys are checked automatically and a session with SSH server is established. There are several options available with **client.connect()**. For example, it's possible to specify certificates using **pkey** or **key_filename** arguments as well as to set user password with **password** argument, if it is not possible to engage better authentication methods.

Therefore, in the worst, insecure and strongly not advised way, the code, as it is shown in Example 7, may even contain passwords in plain text:

```
client.connect(hostname='itso_storage', username='itso_user', password='MySecReT1')
```

Example 7   Dangerous, not recommended **client.connect()** method invocation

To run a command on the remote system **exec_command()** method is used. It returns data from **stdin**, **stdout** and **stderr** streams of the executed command.

In the last lines of the script the data is read from both stdout and stderr streams and decoded into convenient US-ASCII character set to be printed by **print()**.

SMI-S

IBM Spectrum Virtualize supports the Storage Management Initiative Specification (SMI-S). It is an ISO approved storage standard which provides access to common management functions and features of various storage systems. It is developed and maintained by the Storage Networking Industry Association (SNIA).

SMI-S is based on three main components: the Common Information Model (CIM), Web-Based Enterprise Management standards (WBEM) and Service Location Protocol (SLP).

There is a number of applications and tools which can be used to query SMI-S, but this chapter shows a simple way to reach CIM/WBEM services of the IBM SAN Volume Controller system using Python and open-source PyWBEM module, which can be installed via **pip**.

Example 8 demonstrates the very basics of pywbem module usage with IBM SVC system:

```
import pywbem
import getpass

target = 'itso_storage'
url = 'https://' + target
username = 'itso_user'
password = getpass.getpass()

wbemc = pywbem.WBEMConnection(url, (username, password), 'root/ibm',
no_verification=True)

cluster = wbemc.EnumerateInstances('IBMTSSVC_Cluster')
print(cluster[0].items())
```

Example 8   Basic pywbem usage

In Example 8, **WBEMConnection()** establishes HTTPS connection with WBEM services of IBM SAN Volume Controller. Here, target storage system URL is specified by the **url** argument. **Username** and **password** as well as the CIM namespace (**root/ibm**) to query are also provided in the next lines.

Note, **getpass** module is not necessary to work with SMI-S, it's purpose is to securely read passwords from standard input with terminal echo function switched off to hide what is typed in.

Next, **no_verification=True** argument disables SSL certificate verification in this code. In other words it forces the script to trust any certificate provided by the WBEM server.

Warning, disabling the verification of the server SSL certificate is dangerous. It is insecure and should be avoided in production.

After connection is successfully established, instances of a given CIM class can be enumerated with **EnumerateInstances()** method, which returns a complex data structure – a list of **CIMInstance()** classes. In Example 8, it is done over **IBMTSSVC_Cluster** class which represents system level information comparable with the results of **lssystem** command execution.

There are different CIM classes available for comprehensive management of SAN Volume Controller system, such as:

- IBMTSSVC_Cluster – System level information
- IBMTSSVC_Node – Information about nodes
- BMTSSVC_ConcreteStoragePool – Mdisk groups
- IBMTSSVC_BackendVolume – Mdisks themselves
- IBMTSSVC_StorageVolume – Vdisk information
- etc.

This document touches small amount of them to illustrate SMI-S capabilities, but it does not provide full list of these classes or their descriptions. See documentation on IBM SAN Volume Controller WBEM/CIM classes, their purposes and relationship diagrams on this website:
https://www.ibm.com/support/knowledgecenter/STPVGU/com.ibm.storage.svc.console.720.doc/svc_sdkintro_215ebp.html

The last line of the script parses and prints out the data. But it is not the only way to do the job, Python is a flexible language, it allows to do such work in different ways. Several approaches of processing the data acquired by **EnumerateInstances()** for a number of CIM classes are listed in Example 9:

```
print('Cluster information')
cluster = wbemc.EnumerateInstances('IBMTSSVC_Cluster')
print(cluster[0]['ElementName'])
for c_prop in cluster[0]:
    print('\t{prop}: "{val}"'.format(prop=c_prop, val=cluster[0].properties[c_prop].value))

print('Nodes information')
nodes = wbemc.EnumerateInstances('IBMTSSVC_Node')
for node in nodes:
    print(node['ElementName'])
    for n_prop in node:
        print('\t{prop}: "{val}"'.format(prop=n_prop, val=node[n_prop]))
print('Pools information')
pools = wbemc.EnumerateInstances('IBMTSSVC_ConcreteStoragePool')
print('PoolID', 'NumberOfBackendVolumes', 'ExtentSize', 'UsedCapacity',
      'RealCapacity', 'VirtualCapacity', 'TotalManagedSpace', sep=',')
for pool in pools:
    print(
        pool['ElementName'], pool['NumberOfBackendVolumes'], pool['ExtentSize'],
        pool['UsedCapacity'], pool['RealCapacity'], pool['VirtualCapacity'],
        pool['TotalManagedSpace'], sep=','
    )
```

Example 9 Parsing EnumerateInstances() output for various classes representing Cluster, Nodes and Storage pools

Using similar, yet different approaches “Cluster information” and “Nodes information” sections of the example parse data in key/value pairs to show all acquired data. On the other hand, “Pools information” part filters data to print selected fields only. It wastefully ignores all other fields.

For some classes, like **IBMTSSVC_StorageVolume**, full enumeration of all the instances may be quite slow, it can generate tens megabytes of unnecessary data which must be prepared by the storage system, passed over the network and finally parsed by the script. Fortunately, it is possible to significantly reduce such data-flows by requesting limited amount of necessary information only.

There is **ExecQuery()** method which allows to request the WBEM server in a convenient query language, similar to SQL. Two dialects: CIM Query Language (DMTF:CQL) and WBEM Query Language (WQL) are recognized by PyWBEM, both of them can be used with IBM SAN Volume Controller, but this chapter uses CQL syntax in examples. DMTF specification (DSP0202) for CQL can be found on this website:
https://www.dmtf.org/sites/default/files/standards/documents/DSP0202_1.0.0.pdf

Example 10, illustrates flexibility of the **ExecQuery()** method:

```
print('Vdisks:')
vdisks = wbemc.ExecQuery(
    'DMTF:CQL',
    "SELECT VolumeId, VolumeName, NumberOfBlocks FROM IBMTSSVC_StorageVolume"
    " WHERE VolumeName LIKE 'vdisk.'"
)
for vd in vdisks:
    print(vd['VolumeId'], vd['VolumeName'], vd['NumberOfBlocks'], sep=',')
```

Example 10 Querying required only data with **ExecQuery()** method

One of the advantages of SMI-S on IBM SVC is it's capability to collect performance data of various storage system components using “Statistic” family CIM classes. For example:

- IBMTSSVC_BackendVolumeStatistics
- IBMTSSVC_FCPortStatistics
- IBMTSSVC_NodeStatistics
- IBMTSSVC_StorageVolumeStatistics

- etc.

A quite detailed, with commentaries, example of performance data collecting and processing script is shown in Example 11. It works with **IBMTSSVC_StorageVolumeStatistics** to retrieve **vdisks** statistics:

```
import pywbem
import getpass
import time

target = 'itso_storage'
user = 'itso_user'
password = getpass.getpass()

url = 'https://' + target
ofs = ','                             # Output field separator
header = ['InstanceID', 'ReadIOs', 'WriteIOs', 'TotalIOs',
          'KBytesRead', 'KBytesWritten', 'KBytesTransferred']
frequency = 5       # Performance collection interval in minutes


def vdisks_perf(wbem_connection, hdr):
    """Get performance statistics for vdisks"""

    # Form "select" request string
    request = "SELECT " + ','.join(hdr) + " FROM IBMTSSVC_StorageVolumeStatistics"
    result = []

    # Request WBEM
    vd_stats = wbem_connection.ExecQuery('DMTF:CQL', request)
    # parse reply and form a table
    for vds in vd_stats:
        # Handle 'InstanceID' in a specific way
        vde = [int(vds.properties[hdr[0]].value.split()[1])]

        # Collect the rest of numeric performance fields
        for fld in header[1:]:
            vde.append(int(vds.properties[fld].value))
        result.append(vde)

    return result


def count_perf(new, old, interval):
"""Calculate performance delta divided by interval to get per second values"""

    result = []
    for r in range(0, len(new)):
        row = [new[r][0]]                     # InstanceID
        for c in range(1, len(new[0])):
            row.append(round(float(new[r][c] - old[r][c]) / interval, 2))
        result.append(row)
    return result


def print_perf(stats, hdr):
    """Printout performance data matrix"""

    # Print header
    print(ofs.join(str(fld) for fld in hdr))

    # Print performance table
    for ln in stats:
        print('{}{}{}'.format(ln[0], ofs, ofs.join(str(fld) for fld in ln[1:])))

# Connect with WBEM/CIM services
wbemc = pywbem.WBEMConnection(url, (user, password), 'root/ibm', no_verification=True)

# Infinite performance processing loop
new_perf = vdisks_perf(wbemc, header)
while True:
    old_perf = new_perf
    new_perf = vdisks_perf(wbemc, header)
    delta_perf = count_perf(new_perf, old_perf, frequency * 60)

    print_perf(delta_perf, header)
    time.sleep(frequency * 60)
```

Example 11 Accessing performance metrics with PyWBEM module

Note, statistic collection must be already configured and running on the storage system. To check it, execute:

```
lssystem | grep statistics
```

To set appropriate statistic generation interval, run:

```
startstats -interval N
```

Where N is interval in minutes. For Example 11 to work properly, interval must be 5 minutes or less.

SMI-S services can be used not only to report, but to configure storage systems. Algorithms of such operations on the storage system can be accessed by the following links:

https://www.ibm.com/support/knowledgecenter/STPVGU/com.ibm.storage.svc.console.720.doc/svc_storagecfgintro_215fca.html

https://www.ibm.com/support/knowledgecenter/STPVGU/com.ibm.storage.svc.console.720.doc/svc_copyservicesintro_215fcb.html

HTTPS and RESTful API on IBM Spectrum Control

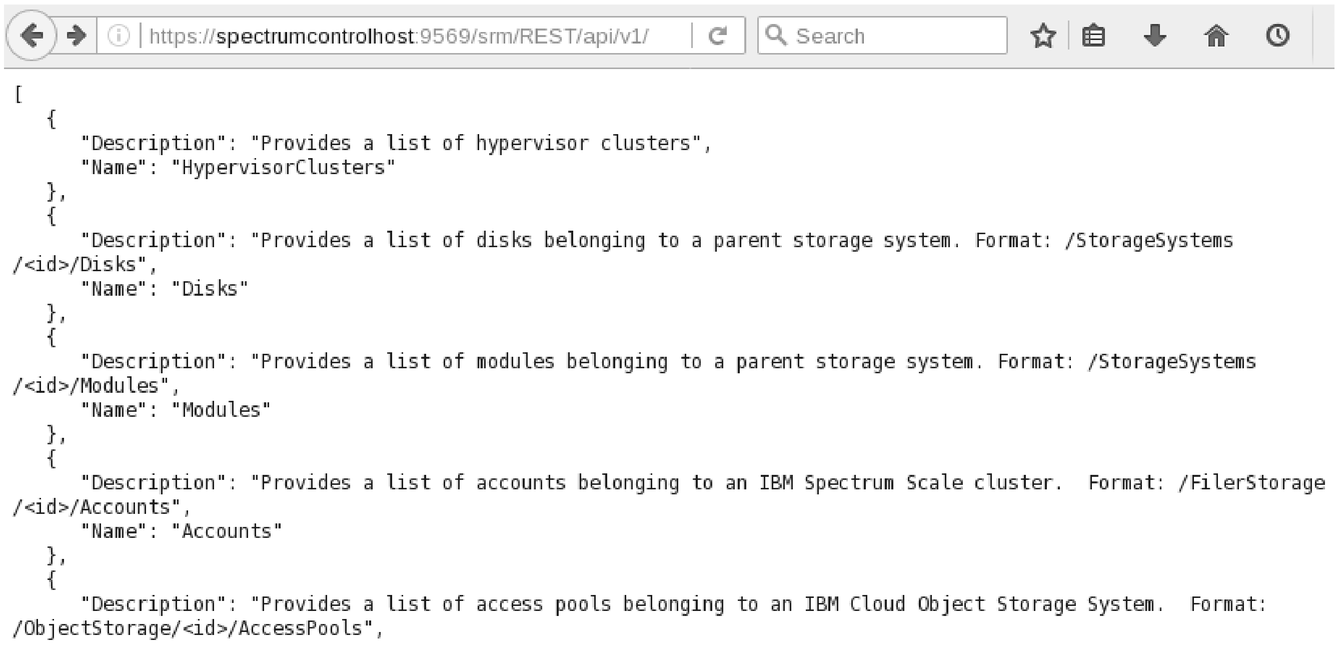
Different approach to access storage resources is to use Hypertext Transfer Protocol Secure (HTTPS) protocol and Representational State Transfer (REST or RESTful) API of IBM Spectrum Control server. The main advantage of this method is that it allows to get information about the whole SAN/Storage infrastructure managed by the IBM Spectrum Control server.

IBM Spectrum Control REST API documentation is available by the link:

https://www.ibm.com/support/knowledgecenter/SS5R93_5.2.15/com.ibm.spectrum.sc.doc/ref_rest_api.html

The basic idea of it is to send an HTTP GET request for a specific URL to retrieve structured information about the storage resource.

For example, requesting https://spectrumcontrolhost:9569/srm/REST/api/v1/ in the browser will show the root of the whole REST tree: access points for all sections of all possible resources registered in the Spectrum Control server:



```
[
   {
      "Description": "Provides a list of hypervisor clusters",
      "Name": "HypervisorClusters"
   },
   {
      "Description": "Provides a list of disks belonging to a parent storage system. Format: /StorageSystems
/<id>/Disks",
      "Name": "Disks"
   },
   {
      "Description": "Provides a list of modules belonging to a parent storage system. Format: /StorageSystems
/<id>/Modules",
      "Name": "Modules"
   },
   {
      "Description": "Provides a list of accounts belonging to an IBM Spectrum Scale cluster.  Format: /FilerStorage
/<id>/Accounts",
      "Name": "Accounts"
   },
   {
      "Description": "Provides a list of access pools belonging to an IBM Cloud Object Storage System.  Format:
/ObjectStorage/<id>/AccessPools",
```

Figure 1 Accessing Spectrum Control RESTful API using HTTP GET request

Table 1 lists the wide range of sections, some of which are nested:

Accessers	NASFileSystems
AccessPools	NASNodes
Accounts	NASNSDs
Applications	NASPools
Blades	NASSnapshots
ClusterNodes	Nodes
Clusters	ObjectStorage
Containers	Performance
Departments	Pools
Disks	RAIDArrays
Fabric	Relationships
FilerStorage	RemoteReplication
GeneralGroups	Servers
HostConnections	Sites
HypervisorClusters	Slicestors

Hypervisors	StoragePools
InterSwitchConnections	StorageSystems
IOGroups	Switches
ManagedDisks	SwitchPorts
Mirrors	Vaults
Modules	Volumes
NASFileSets	ZoneSets

Table 1   Branches and leaves of the Spectrum Control RESTful API tree

It is possible to access any of these sections manually with a browser, command-line **curl** or **wget** web-retrieving utilities. Example of using **wget** is shown on the web-site:

https://www.ibm.com/support/knowledgecenter/SS5R93_5.2.15/com.ibm.spectrum.sc.doc/mgr_rest_api_retrieving_cli.html

It is also easy to automate the same operations using Python as the output is produced in the well-known JavaScript Object Notation (JSON) format.

Example 12 demonstrates the code to fetch information about storage systems. To simplify the work with HTTPS, open-source module **requests** is used to perform the task:

```
import requests
import getpass

username = 'scuser'
password = getpass.getpass()

url = 'https://spectrumcontrolhost:9569/srm/'

sess = requests.Session()
sess.verify = False

resp = sess.post(url + 'j_security_check',
                 data={'j_username': username, 'j_password': password})
resp.raise_for_status()

resp = sess.get(url + 'REST/api/v1/' + 'StorageSystems')
resp.raise_for_status()

print(resp.json())
```

Example 12 Accessing Spectrum Control RESTful API using Python and **requests** module

To request password interactively and to hide user's typing “getpass” module is engaged.

To log a user in Spectrum Control authentication procedure requires retrieving a security token from the server and storing it to use during the session. Therefore, to preserve cookies (including the security token) and reuse the same TCP connection with the server (for performance benefits) **requests.Session()** is initialized in the script: **sess = requests.Session()**. As in the examples for WBEM/CIM (see Example 8 and Example 11) and because of the same reasons, SSL certificate verification here is also disabled: **sess.verify = False**.

Actual requests are sent by sess.post() and sess.get() within the same session. In the first case, HTTP POST is used to submit username and password via login form at:

https://spectrumcontrolhost:9569/srm/j_security_check

The second one is an ordinary HTTP GET to request RESTful API for storage systems information: https://spectrumcontrolhost:9569/srm/REST/api/v1/StorageSystems

In the script **raise_for_status()** is executed on every response. It raises an exception if the server returns 4xx or 5xx error status code but passes transparently if request was successful.

The last line of the code in Example 12, decodes JSON data of the response into convenient Python structure of nested lists and dictionaries. On this step any appropriate actions can be applied to parse, filter and process acquired data, but for the sake of simplicity, the script just prints this structure to standard output.

HTTPS on IBM Spectrum Virtualize

Though it is less documented, it is possible to interact with Hypertext Transfer Protocol Secure (HTTPS) server of IBM Spectrum Virtualize directly the very same way typical browsers do. And as it was shown in the previous examples for Spectrum Control, Python with open-source module **requests** can greatly simplify the task.

Example 13 contains basic code of establishing HTTPS connection with IBM Spectrum Virtualize system to request and process information about virtual disks:

```
import getpass
import requests
import time

target = 'itso_storage'
user = 'itso_user'
password = getpass.getpass()

target_url = 'https://' + target + '{}'

sess = requests.Session()
sess.verify = False

print('Accessing the system')
resp = sess.get(target_url.format('/'))
resp.raise_for_status()

time.sleep(1)
print('Trying to login')
resp = sess.post(target_url.format('/login'),
                 data={'login': user, 'password': password})
resp.raise_for_status()
```

```
print('Requesting data')
resp = sess.post(target_url.format('/VdiskGridDataHandler'),
                 data={'sort': 'vdiskUid'})
resp.raise_for_status()

vdisks = resp.json()
for vdisk in vdisks['items']:
    print(vdisk['name'], vdisk['capacity'], vdisk['vdiskUid'],
          vdisk['mdiskGrpName'], vdisk['status'])
```

Example 13 Using HTTPS to query IBM Spectrum Virtualize systems

Besides **requests**, two other modules are imported in this code: **getpass** is needed to hide user password, and **time** is used to make a short pause between requests.

A session with disabled SSL certificate verification is established like it was shown and described in Example 12.

Before requesting the server for storage related information, like vdisks data in this example, two steps must be completed in advance:

1. A valid session cookies must be acquired from the server. In Example 13 it is performed with simple GET request to the root of the server: **sess.get(target_url.format('/'))**. Note, a small pause **time.sleep(1)** after this step may be required for the server to set up an environment for the new session.
2. A user must be authenticated by the storage system. It is done by filling in login form with POST request: **sess.post(target_url.format('/login'), data={'login': user, 'password': password})**

Only after successful accomplishing both of these stages a request to query an actual storage related information can be done.

As it was done earlier in Example 12, **resp.raise_for_status()** is called to catch HTTP errors if they occur.

The code above accesses **/VdiskGridDataHandler** recourse, but actually any valid page available via WEBUI can be requested from the server, for example: **/FlashCopyTreeDataHandler**, **/HostGridDataHandler**, etc. Another interesting URL is **/RPCAdapter** which can be used to obtain performance statistics from the storage system.

The HTTP POST method to query such resources is needed to submit additional parameters in JSON format or various form options, such as sorting order **{'sort': 'vdiskUid'}** shown in Example 13.

As the server also responds in JSON it is useful to process it with built-it **requests** module JSON parser: **vdisks = resp.json()**.

To keep the example simple, the last lines of the script define just a **for**-loop to go over the acquired vdisks information and print some of the valuable items.

In more recent versions like 7.8.1.6 and above, two new variables **kzgjfybjm** and **kzgjfycsm** have been added. They are used to control the session on a "handshake" manner: the server sets **kzgjfybjm** cookie and the client must add it's value into **kzgjfycsm** variable in the request URL. Note, **kzgjfybjm** and **kzgjfycsm** values change from time to time, so it's good to add fresh values after getting every response from the server. So the script above can be changed:

```
import requests
import time

target = 'IBM_SVC_Storage_system'
user = 'My_user_name'
password = 'My_secret_password'
target_url = 'https://' + target + '{}'

sess = requests.Session()
sess.verify = False

print('Accessing the system')
resp = sess.get(target_url.format('/'))
resp.raise_for_status()

time.sleep(1)
print('Trying to login')
resp = sess.post(target_url.format('/login'), data={'login': user, 'password': password})
resp.raise_for_status()

handshake = 'kzgjfycsm' + '=' + resp.cookies['kzgjfybjm']

print('Getting data')
resp = sess.post(target_url.format('/VDiskGridDataHandler?' + handshake),
                 data={'count': '40', 'extendedMDiskInfo': 'false', 'refresh': 'true', 'start': '0'})
resp.raise_for_status()

# Print out information:
print(resp.json())

# Note! Before sending a new request, refresh 'handshake' again:
# handshake = 'kzgjfycsm' + '=' + resp.cookies['kzgjfybjm']
```

Example 14 Using HTTPS to query IBM Spectrum Virtualize systems – updated

Conclusions

This chapter briefly showed several unified methods of accessing IBM SVC/Storwize systems with scripts. Modern and open protocols, APIs and standards allow to write versatile code for storage management, reporting and configuration tasks using programming languages convenient for different environments and operating systems.

SNMP protocol on IBM Spectrum Virtualize is primary used for system monitoring tasks. It is not touched by the scripting topic because SNMP-agents are configured to work in send traps only mode and it is not possible to access them with SNMP GetRequest/SetRequest/etc requests.