XENA NETWORKS

USER MANUAL

Valkyrie REST Server

User Manual

This document describes how to build client applications
for Xena Valkyrie REST server.

VALKYRIE
LAYER 2-3 ETHERNET TRAFFIC GENERATION AND ANALYSIS SOLUTIONS

Last updated: 2018-09-27

CONTENTS

GENERAL

This document describes how to build client applications for Xena Valkyrie REST server (hereafter – REST server or simply server). The document contains two parts:

- Step by step implementation of basic work flow
- Summary and general architecture notes

FUNCTIONALITY

The REST server goal is to replace the CLI API as modern best practice method for automation.

As such, it follows the CLI commands structure and maps each CLI command to a REST URL.

Each CLI commands family (chassis, port, stream etc.) is mapped to REST namespace.

The indices and sub-indices of the specific command are mapped to the URL.

For example, all stream commands have two indices module/port and one sub-index [stream]. In REST it will look like:

/session/{user}/chassis/{chassis-address}/module/{module}/port/{port}/stream/{stream}

As you can see, in addition to module/port/stream indices we also have session ID and chassis address. These additional indices are required as REST server is multi-user and multi-chassis.

AUDIENCE AND PREREQUISITES

The document assumes the reader is familiar with Valkyrie CLI API and REST basics. To run the sample flow the user should have access to Valkyrie chassis with two available ports. It is recommended the ports be connected back-to-back so sent traffic will arrive the receive port but this is not mandatory.

HOW TO READ THIS DOCUMENT

We start by describing the general structure of the REST routes and commands.

Then we have a step-by-step sample script that shows and explains the concepts of:

- Create session
- Reserve two ports
- Create stream
- Run traffic
- Get statistics

Throughout the flow we will explain the structure and functionality of the REST server by example, using CURL commands. In Appendix A you can find short intro to curl commands and the list of all commands used throughout the example.

General concepts are colored green.

You can skip the general part and go directly to the sample.

If you are familiar enough with REST concepts you can download the swagger.json and skip directly to Appendix A to play around with the curl commands of the sample script.

INSTALLING AND GETTING STARTED WITH XENA VALKYRIE REST SERVER

Starting from Xena Valkyrie software release 77 the chassis supports REST Server in addition to the CLI API.

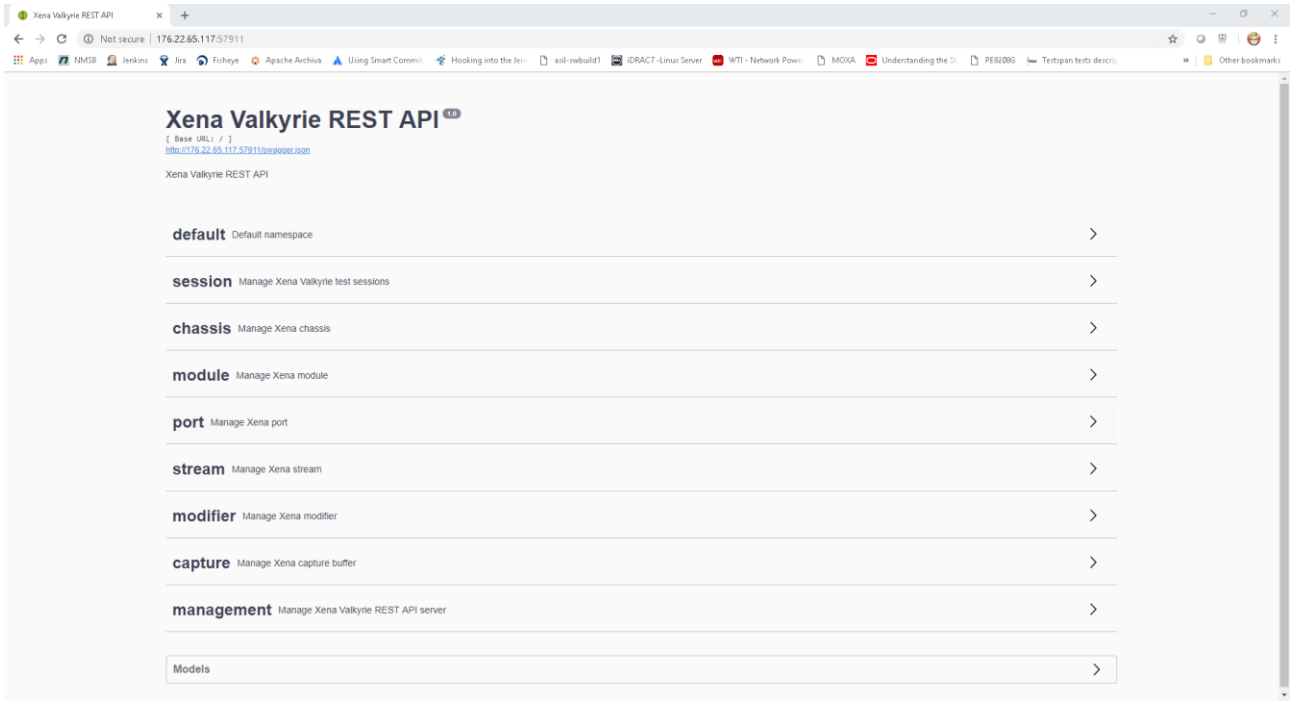
The REST server is enabled and running on port 57911 by default. No need to install or start anything. Just go to URL http://chassis IP:57911 and start your REST session.

SWAGGER UI

To get to the starting point of Xena REST server just go to:

http://Chassis IP Address:57911

If you set this URL in a browser, you will get to the swagger UI:



Here you can see all namespaces and try out all REST commands to get to know the API better.

If you browse to http://Chassis IP Address:57911/swagger.json you can get the swagger file behind the server.

EXISTING CLIENTS

The latest version of xenavalkyrie python package supports REST API. You can download it from pypi.

REST URLS AND COMMANDS

The URL tree is composed of three URL types – Objects Family, Specific Object and Operations.

The basic structure is:

- **Family URL**
  - **Specific object URL**
    - **Operations URL**

Each URL supports different REST commands.

The description below is high level only. To get the complete and REST definitions download the swagger JSON file from http://rest-server-ip:57911/swagger.json.

OBJECTS FAMILY URL

Objects family URLs represent an object family like sessions, chassis, modules, ports, streams etc.

URL objects have the general form of URL-of-specific-object/family-name.

Some examples:

- The URL of the sessions family is /session
- The URL of chassis list under "foo" session is /session/foo/chassis.
- The URL of streams under localhost port 0/0 is /session/foo/chassis/localhost/module/0/port/0/stream

Objects family URLs support GET and [POST] commands:

- GET

  Return the list of identifiers of the family objects.

  Some examples:

  - GET /session returns list of sessions
  - GET /session/foo/chassis will return the chassis list of "foo" session
  - GET /session/foo/chassis/localhost/module/0/port/0/stream will return the list of streams under localhost chassis port 0/0.
- POST

  Creates new object of the family.

  Returns the new object ID, except for session and chassis.

  Some samples:
  - POST /session?user=foo creates session named foo.
  - POST /session/foo/chassis?ip=192.168.1.195 will connect to chassis 192.168.1.195.
  - POST /session/foo/chassis/localhost/module/0/port/0/stream will create new stream under localhost port 0/0 and return it's ID.

SPECIFIC OBJECT URL

Specific object URLs represent a single object like session, chassis, module, port, stream etc.

URL objects have the general form of URL-of-family/specific-object-id.

Some examples:

- The URL of the “foo” session is /session/foo
- The URL of localhost chassis under “foo” session is /session/foo/chassis/localhost
- The URL of localhost port 0/0 stream 0 is /session/foo/chassis/localhost/module/0/port/0/stream/0

Specific object URLs support GET and [DELETE] commands:

- GET
  Return all sub-routes of the specific object URL. These include object operations sub-routes and object family children sub-routes.
- DELETE
  Deletes the object.
  Some examples:
  - DELETE /session/foo/chassis/localhost/module/0/port/0/stream/0 will delete stream 0 under localhost port 0/0.
  - DELETE /session/foo will disconnect from all chassis and delete session “foo”.

OPERATIONS URL

All CLI commands can be divided to three types:

- Commands – commands that operate on an object.
- Attributes – commands that represent single attribute of an object.
- Statistics – commands that returns object statistics.

Operations URLs allow you to perform operations on the specific parent object. There are three standard operations routes, one for each CLI command type – commands, attributes and statistics. Some objects support specific operations like chassis/backdoor or port/files.

Commands URL

Commands URL allows you to perform any raw CLI command on the object.

Some examples:

- The URL to access commands on localhost chassis under “foo” session is /session/foo/chassis/localhost/commands
- The URL to access commands on localhost port 0/0 stream 0 is /session/foo/chassis/localhost/module/0/port/0/stream/0/commands

Commands URL supports GET command.

- GET
  Returns a list of all CLI commands that can be performed on the object.
  Note that this command is not implemented in this release and will always return 501.

Specific Command URL

Command URL allows you to perform a raw CLI command on the specific object. When performing command from the commands URL the CLI command parameters are transferred as a list of strings and returned value type must be specified. The return value is the CLI command return value as a raw string.

Specific command URL supports only the POST commands.

- POST

  Some examples:

  - The URL to reserve port localhost 0/0 is
    /session/foo/chassis/localhost/module/0/port/0/commands/p_reservation
    with body parameter
    {"parameters": ["reserve"],
    "return_type": "no_output"}
    ➔ null
  - The URL to get reservation status of port localhost 0/0 is
    /session/foo/chassis/localhost/module/0/port/0/commands/p_reservedby
    with body parameter
    {"parameters": ["?"],
    "return_type": "line_output"}
    ➔ "\"foo\""
  - The URL to get total TX total statistics of port localhost 0/0 is
    /session/foo/chassis/localhost/module/0/port/0/commands/pt_total
    with body parameter
    {"parameters": ["?"],
    "return_type": "multiline_output"}
    ➔ ["\n", "0/0 PT_TOTAL 0 0 384000 6000\n"]

Note that while All CLI commands can be performed from the commands URL it is easier to access attributes from the attributes URL and statistics from the statistics URL as shown below.

Attributes URL

Attributes URL allows you to access – get and set – object attributes.

Some examples:

- The URL to access attributes on localhost chassis under “foo” session is
  /session/foo/chassis/localhost/attributes
- The URL to access attributes on localhost port 0/0 stream 0 is
  /session/foo/chassis/localhost/module/0/port/0/stream/0/attributes

Attributes URL supports GET and PATCH commands.

- GET
  Returns a dictionary of all object attributes with their current value.
- PATH
  Sets a list of attributes for the specific object. The list of attributes to set with their values is transferred as body parameters.

Some example:

- The URL to set attributes localhost port 0/0
  PATH /session/foo/chassis/localhost/module/0/port/0/attribute
  with body parameter
  [{"name": "p_txenable", "value": "OFF"},
  {"name": "p_autonegselection", "value": "ON"}]

Statistics URL

Statistics URL allows you to retrieve the statistics of the specific object.

Some examples:

- The URL to retrieve statistics on port localhost 0/0 is
  /session/foo/chassis/localhost/module/0/port/0

Statistics URL supports GET command only.

- GET

  Returns a dictionary of all object statistics.

Backdoor URL

The backdoor URL is special chassis sub-route. It allows you to perform any CLI command on the chassis, returning the raw string output of the CLI command. This is the last resort backdoor for any missing functionality of the REST server.

When performing command from the backdoor URL the CLI command along with its parameters are transferred as single string and returned value type must be specified. The return value is the CLI command raw output as a string.

Some examples:

- The URL to perform any CLI command on localhost under “foo” session is
  /session/foo/chassis/localhost/backdoor

The backdoor URL supports POST operation only.

- POST
  Performs any CLI command on the chassis.
  Some examples:
  - The URL to get reservation status of port localhost 0/0 from backdoor is
    /session/foo/chassis/localhost/backdoor
    with body parameter
    {"command": "0/0 p_reservedby ?",
    "return_type": "line_output"}
    ➔ "0/0 P_RESERVEDBY \"foo\""

Files URL

The files URL is special port sub-route. It is used to access – load and save – port configuration files.

Some examples:

- The URL to access configuration files of port localhost 0/0 is /session/foo/chassis/localhost/module/0/port/0/files

The files URL supports GET and POST commands.

- GET
  Returns the port configuration file as a multiple string.
  Some examples:
  - The URL to download configuration of port localhost 0/0 is http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/files
- POST
  Uploads port configuration file.
  Some examples:
  - The URL to upload c:/temp/curl_config.xpc file onto port localhost 0/0 is http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/files with file parameter "file=@c:/temp/curl_config.xpc"

BASIC AUTOMATION FLOW

CREATE SESSION AND CONNECT TO CHASSIS

A REST flow starts with a session.

The root URL for the REST server is /session.

To create a session use POST /session command with the session (==user) name.

Let's create two sessions – foo and bar.

curl -X POST http://176.22.65.117:57911/session?user=foo

curl -X POST http://176.22.65.117:57911/session?user=bar

Creating new objects (session, stream, modifier…) is performed with POST command on the parent object URL

Wherever possible we transfer parameters as QUERY parameters as this is the simplest form,

Now we can use GET /session to get a list of all active sessions:

curl -X GET http://176.22.65.117:57911/session

➔ *{"routes": [],*
"sessions": [{"user": "foo"}, {"user": "bar"}]}

GET /session returns two lists:

- Routes – list of all sub-routes of /session URL. Sessions family URL has no sub-routes.
- Sessions – list of all sessions. Each session is represented by directory holding its information. In this version the only information is the user name hence there is only one entry per session.

REST server always returns object information in dictionary. This allows us to add new information as new fields in future versions while maintaining backward compatibility.

To manage specific session we should simply use its name in the URL - /session/{name}

It's always good to start with GET and OPTIONS to see what are the sub-routes and commands available for the URL:

curl -X GET http://176.22.65.117:57911/session/foo

➔ *{"routes": [{"name": "statistics", "description": "REST session statistics"}*
{"name": "chassis", "description": "Manage chassis list"}],
"objects": []}

GET /session/foo returns two lists:

- Routes – list of all sub-routes of /session/foo URL. Session URL has two sub-routes:
  - Chassis – manage chassis.
  - Statistics – REST session statistics.

- Objects – list of child objects. Session has no child-objects.

Once we create a session the REST server will add the chassis to the chassis list as “localhost”. So if we perform GET /session/<name>/chassis we see the following chassis list:

curl -X GET http://176.22.65.117:57911/session/foo/chassis

➔ *{"routes": [],*
"chassis": [{"ip": "localhost", "password": null, "port": null}]}

GET /session/foo/chassis returns two lists:

- Routes – list of all sub-routes of chassis URL. Chassis family URL has no sub-routes.
- Chassis – list of all chassis. Each chassis is represented by directory holding its information – IP, port and password. In this version the REST server does not retrieve the port/password from the chassis thus it returns default “null” values.

The null value indicates Unsupported (the REST server does not retrieve the actual values).

The localhost can be accessed also using 127.0.0.1 and its actual IP. We will demonstrate it in the next section.

Let’s delete bar as we don’t really need it:

DELETE http://176.22.65.117:57911/session/bar

Deleting objects (sessions, streams, modifiers…) is perform with DELETE command on the object URL

And if we GET /session again:

curl -X GET http://176.22.65.117:57911/session

➔ {"routes": [],
"sessions": [{"user": "foo"}]}

GET INVENTORY

Once we have session and chassis we can get the inventory and reserve ports for the test.

If we want to see the modules we should simply GET /session/<name>/chassis/<ip>:

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost

➔ *{"routes": [{"name": "attributes", "description": "Chassis attributes"},*
{"name": "commands", "description": "Chassis commands"},
{"name": "statistics", "description": "Chassis statistics"},
{"name": "module", "description": "Manage modules"},
{"name": "backdoor", "description": "Chassis back-door to execute any CLI command"}],
"objects": []}

As said above, you can access the localhost also with 127.0.0.1 or it’s actual IP.

curl -X GET http://176.22.65.117:57911/session/foo/chassis/127.0.0.1

curl -X GET http://176.22.65.117:57911/session/foo/chassis/176.22.65.117

➔ *{"routes": [{"name": "attributes", "description": "Chassis attributes"},*
{"name": "commands", "description": "Chassis commands"},
{"name": "statistics", "description": "Chassis statistics"},
{"name": "module", "description": "Manage modules"},
{"name": "backdoor", "description": "Chassis back-door to execute any CLI command"}],
"objects": []}

Now it's time to get all modules. From the response to the GET /session/foo/chassis/localhost we learn that it has sub-route named "module" that is used to manage its modules.

So we GET /session/foo/chassis/localhost/module:

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module

➔ *{"routes": [],*
"objects": [{"id": 0, "name": null}]}

And learn that localhost chassis has one module at slot zero.

And then we move to module zero:

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module/*0*

➔ *{"routes": [{"name": "attributes", "description": "Get/Set module attributes"},*
{"name": "commands", "description": "Module commands"},
{"name": "port", "description": "Manage ports"}]
"objects": []}

Again, from the response to the GET /session/foo/chassis/localhost/module/0 we learn that it has sub-route named "port" that is used to manage its ports.

So we GET /session/foo/chassis/localhost/module/0/port:

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port

➔ *{"routes": [],*
"objects": [{"id": 0, "name": null}, {"id": 1, "name": null}]}

And learn that localhost chassis module zero has ports – zero and one.

Let's see what\s under port zero then we can generalize.

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module/*0/port/0*

➔ *{"routes": [{"name": "files", "description": "Port configuration files"}*
{"name": "capture", "description": "Manage capture buffer"},
{"name": "commands", "description": "Port commands"},
{"name": "statistics", "description": "Port statistics"},
{"name": "stream", "description": "Manage streams"},
{"name": "tpld", "description": "Manage TPLDs"},
{"name": "attributes", "description": "Port attributes"}],
"objects": []}

Now let's generalize…

The REST objects tree has two node types – objects family node (session, chassis, module and port) and single object nodes (session foo, chassis localhost, module 0, port 0 and port 1).

When you GET a node it will return its sub-routes and child objects. In the current version object family nodes do not have sub-routes so they return only children while specific object nodes do not have children and return only sub-routes.

So when you GET a family node you get the objects in the family and when you GET a specific object you get its sub-routes which will allow you to act on the object or drill down to its children families.

Each object has at least two actions sub-routes:

- Commands – run any CLI command on the object.
- Attributes – get/set any CLI attribute

Many objects have a third sub-route:

- Statistics – get object statistics

And some objects have special action sub-routes like chassis/backdoor and port/files.

Reserve Ports

Now it's time to reserve the ports.

The CLI commands that handles port reservation are:

P_RESERVATION whatdoto – to perform the action of port reservation.

P_RESERVEDBY username – to find out the current port reservation status.

Note that the two commands are different by nature. P_RESERVEDBY is an attribute command that allows you to access an attribute. P_RESERVATION is an operational command that allows you to perform an action.

All CLI commands can be divided to three types:

- Actions – commands that operate on an object.
- Attributes – commands that represent single attribute of an object.
- Statistics – commands that returns object statistics.

All CLI commands can be performed from the commands sub-route. However, it is better to access attributes from the attributes sub-route and statistics from the statistics sub-route.

When performing command from the commands sub-route the CLI command parameters are transferred as a list of strings and returned value type must be specified. The return value is the CLI command return value as a raw string. This is good for operational commands that are less structured. However, attributes and statistics are more structured and the attributes and statistics take advantage of this structure.

First, let's reserve the port:

To reserve the ports we need to POST the P_RESERVATION command URL /session/<name>/chassis/<ip>/module/<module_num>/port/<port_num>/commands/p_reservation with the “reserve” parameter.

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/commands/p_reservation -H "content-type: application/json" -d "{\"parameters\": [\"reserve\"], \"return_type\": \"no_output \"}"

Now let’s see the difference between getting attribute value by commands sub-route and attributes sub-route using the P_ *RESERVEDBY* attribute.

To get the current port ownership status using commands sub-route we need to POST /session/<name>/chassis/<ip>/module/<module_num>/port/<port_num>/commands/p_reservedby with REST body describing the command parameter “?” and the expected output “single_line”:

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/commands/p_reservedby -H "content-type: application/json" -d "{\"parameters\": [\"?\"], \"return_type\": \"single_line\"}"

➔ *["0/0 P_RESERVEDBY \"foo\"\n"]*

As you can see, the returned value is raw stream that requires processing in order to retrieve the actual returned value which is “”.

On the other hand we can use the attributes sub-route to get all attributes as dictionary

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/attributes

➔ *[{"name": "p_reservedby", "value": "foo", "description": null},*
{"name": "p_pause", "value": "OFF", "description": null},
…
{"name": "p_pfcenable", "value": "OFF OFF OFF OFF OFF OFF OFF OFF", "description": null},
{"name": "p_latencyoffset", "value": "0", "description": null}]

And the output is standard list of dictionaries json and the requested value can be extracted with standard json tools.

Let’s reserve port one as well:

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/1/commands/p_reservation -H "content-type: application/json" -d "{\"parameters\": [\"reserve\"], \"return_type\": \"no_output \"}"

RESET AND CONFIGURE PORT

To reset the ports we need to POST the P_RESET command URL /session/<name>/chassis/<ip>/module/<module_num>/port/<port_num>/commands/p_reset with no parameters.

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/commands/p_reset -H "content-type: application/json" -d "{\"parameters\": [], \"return_type\": \"no_output \"}"

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/1/commands/p_reset -H "content-type: application/json" -d "{\"parameters\": [], \"return_type\": \"no_output \"}"

The fastest way to configure a port is to load a configuration file. In Appendix B you can find a simple configuration file with two streams. Copy the commands into a simple txt file and save it as xpc file.

To load a configuration file onto a port we need to POST /session/<name>/chassis/<ip>/module/<module_num>/port/<port_num>/files

curl -F "file=@c:/temp/curl_config.xpc" http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/files

And if we GET /session/<name>/chassis/<ip>/module/<module_num>/port/<port_num>/stream

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/stream

➔ *{"routes": [], "objects": [{"id": 0, "name": null}, {"id": 1, "name": null}]}*

We see that indeed there are two streams on the port.

Run Traffic and Retrieve Statistics

To start traffic we simply POST the P_TRAFFIC command URL /session/<name>/chassis/<ip>/module/<module_num>/port/<port_num>/commands/p_reset with the ON parameter.

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/commands/p_traffic -H "content-type: application/json" -d "{\"parameters\": [\"on\"], \"return_type\": \"no_output \"}"

And finally we can retrieve statistics with GET /statistics URL. Let's get the statistics for stream localhost 0/0/0/

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/stream/0/statistics

➔ *[{"name": "pt_stream",*
"counters": [{"name": "pps", "value": 0},
{"name": "packets", "value": 8000},
{"name": "bps", "value": 0},
{"name": "bytes", "value": 512000}]}]

MANGING THE REST SERVER

CLI COMMANDS

The REST server port and running status (start, stop, restart, enable, disable) can be managed from the CLI.

To learn more about the new CLI command visit the L2-3 Xena Scripting User Manual: https://xenanetworks.com/xena-layer-2-3-scripting-api/.

MANAGEMENT URL

The management URL allows you to manage server attributes, get statistics and perform some basic operations on the server itself.

The management URL is /management.

The structure of the management URL follows the structure of the session URL.

The management URL supports GET command only.

- GET
  Return all sub-routes of the management URL.
  Example:
  - curl -X GET http://176.22.65.117:57911/management
    ➔ {"routes": [
    {"name": "operations", "description": "REST server management operation"},
    {"name": "attributes", "description": "REST server management attributes"},
    {"name": "statistics", "description": "REST server management statistics"},
    {"name": "files", "description": "REST server management logs"}],
    "objects": []}

Attributes URL

Attributes URL allows you to access – get and set – management attributes.

Attributes URL supports GET and PATCH commands.

- GET
  Returns a dictionary of all object attributes with their current value.
  Example:
  - curl -X GET http://176.22.65.117:57911/management/attributes
    ➔ [{"name": "debug", "value": "True", "description": null},
    {"name": "version", "value": "0.6.1", "description": null},
    {"name": "timeout", "value": "360", "description": null},
    {"name": "sessions", "value": "foo", "description": null}]
- PATH
  Sets a list of management attributes. The list of attributes to set with their values is transferred as body parameters.
  Example:

- curl -X PATCH http://176.22.65.117:57911/management/attributes -H "content-type: application/json" -d "[{\"name\": \"debug\", \"value\": \"False\"}]"

Operations URL

Commands URL allows you to perform management commands.

Commands URL supports the GET command only.

- GET
  Returns a list of all management operations that can be performed on the server.
  Example:
  - curl -X GET http://176.22.65.117:57911/management/operations
    ➔ [{"name": "restart", "parameters": null, "description": "restart service"},
    {"name": "reset", "parameters": null, "description": "remove all sessions"},
    {"name": "reserve", "parameters": null, "description": "reserve list of ports under user"}]
    Note that reset and reserve operations are debug operations and will show only when the debug variable is set to True.

Specific Operation URL

Command URL allows you to perform specific management operation. The operation parameters are transferred as a list of strings. Management operations has no return value.

Specific operation URL supports the POST command only.

- POST
  Performs a management operation.
  Example:
  - curl -X POST http://176.22.65.117:57911/management/operations/restart -H "content-type: application/json" -d "{ \"parameters\": []}"

Statistics URL

Statistics URL allows you to retrieve the statistics of the REST server.

Statistics URL supports GET command only.

- GET
  Returns a dictionary of REST server statistics.
  Example:
  - curl -X GET http://176.22.65.117:57911/management/statistics
    ➔ [{"name": "c_reststats",
    "counters": [{"name": "num_sessions", "value": 0},
    {"name": "start_time", "value": 1537780398},
    {"name": "current_time", "value": 1537783877}]}]

Files URL

Files URL allows you to get the log files of the REST server.

The files URL supports GET command only.

- GET
  Returns the REST server log files as zip file.
  Example:
  curl -X GET http://176.22.65.117:57911/management/files

SESSION MANAGEMENT

In order to avoid zombie sessions each session has a timeout. If the server does not get any request from the session client within the timeout period the session will be deleted.

The default timeout is 3600 seconds (== on hour) and it can be set using the /management/attributes URL.

APPENDIX A – SCRIPT CURL COMMANDS

SHORT INTRO TO CURL

This is not a formal documentation of cURL but some basic intro explaining how we use cURL in this document so it is very simplified. For formal introduction and tutorials of CURL just google and you will find plenty of resources.

cURL is a solid and simple tool that allows transferring data from and to any server with command line using various protocols including HTTP. We are using it to send REST commands – GET, POST, PATCH etc.

The general format of a cURL command in this documentation is as follows:

curl –X [GET|POST|PATCH|DELETE] http://chassis-ip:57911/session/...” [*-H "content-type: application/json"] [-d “{}”]*

Where:

- -X is the REST command to perform
- -H is header argument specifying how to read the data if exist. In our case it is always JSON so it will always be *-H "content-type: application/json"*
- -d is the data for POST commands, always JSON.
- -F file location for files upload.

LIST OF SAMPLE COMMANDS

curl -X POST http://176.22.65.117:57911/session?user=foo

curl -X GET http://176.22.65.117:57911/session

curl -X GET http://176.22.65.117:57911/session/foo

curl -X GET http://176.22.65.117:57911/session/foo/chassis

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module/0

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/commands/p_reservation -H "content-type: application/json" -d "{\"parameters\": [\"reserve\"], \"return_type\": \"no_output \"}"

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/1/commands/p_reservation -H "content-type: application/json" -d "{\"parameters\": [\"reserve\"], \"return_type\": \"no_output \"}"

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/commands/p_reset H "content-type: application/json" -d "{\"parameters\": [], \"return_type\": \"no_output \"}"

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/1/commands/p_reset -H "content-type: application/json" -d "{\"parameters\": [], \"return_type\": \"no_output \"}"

curl –X POST -F "file=@c:/temp/curl_config.xpc" http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/files

curl -X POST http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/commands/p_traffic -H "content-type: application/json" -d "{\"parameters\": [\"on\"], \"return_type\": \"no_output \"}"

curl -X GET http://176.22.65.117:57911/session/foo/chassis/localhost/module/0/port/0/stream/0/statistics

APPENDIX B – PORT CONFIGURATION FILE

P_RESET

P_AUTONEGSELECTION ON

P_MDIXMODE AUTO

P_SPEEDSELECTION 255

P_COMMENT "Port 1"

P_SPEEDREDUCTION -1

P_INTERFRAMEGAP 20

P_MACADDRESS 0x04F4BC0E2F64

P_IPADDRESS 0.0.0.0 0.0.0.0 0.0.0.0 0.0.0.0

P_MULTICAST 0.0.0.0 OFF 25

P_MULTICASTEXT 0.0.0.0 OFF 25 IGMPV2

P_MCSRCLIST 0.0.0.0

P_ARPREPLY OFF

P_PINGREPLY OFF

P_IPV6ADDRESS 0x00000000000000000000000000000000 0x00000000000000000000000000000000 128 128

P_ARPV6REPLY OFF

P_PINGV6REPLY OFF

P_ARPRXTABLE

P_NDPRXTABLE

P_PAUSE OFF

P_PFCENABLE OFF OFF OFF OFF OFF OFF OFF OFF

P_RANDOMSEED 0

P_LATENCYOFFSET 0

P_LATENCYMODE LAST2LAST

P_FLASH OFF

P_TXENABLE ON

P_TXTIMELIMIT 0

P_TXMODE NORMAL

P_MAXHEADERLENGTH 128

P_AUTOTRAIN 0

P_LOOPBACK NONE

P_CHECKSUM OFF

P_GAPMONITOR 0 0

P_MIXWEIGHTS 0 0 0 0 57 3 5 1 2 5 1 4 4 18 0 0

P_TXDELAY 0

P_TPLDMODE NORMAL

P_DYNAMIC OFF

P_PAYLOADMODE NORMAL

PS_INDICES 0 1

PS_ENABLE [0] ON

PS_PACKETLIMIT [0] 8000

PS_COMMENT [0] "Stream 1-1"

PS_RATEPPS [0] 1000

PS_BURST [0] -1 100

PS_HEADERPROTOCOL [0] ETHERNET VLAN IP

PS_PACKETHEADER [0]
0x22222222221111111111111181000011080045000002A000000007FFF34D10101010102020201

PS_MODIFIERCOUNT [0] 1

PS_MODIFIER [0,0] 4 0xFFFF0000 INC 1

PS_MODIFIERRANGE [0,0] 0 1 65535

PS_PACKETLENGTH [0] FIXED 64 1518

PS_PAYLOAD [0] INCREMENTING 0x00

PS_TPLDID [0] 0

PS_INSERTFCS [0] ON

PS_IPV4GATEWAY [0] 0.0.0.0

PS_IPV6GATEWAY [0] 0x00000000000000000000000000000000

PS_ENABLE [1] ON

PS_PACKETLIMIT [1] 8000

PS_COMMENT [1] "Stream 1-2"

PS_RATEPPS [1] 1000

PS_BURST [1] -1 100

PS_HEADERPROTOCOL [1] ETHERNET VLAN VLAN IPV6

PS_PACKETHEADER [1] 0x22222222222211111111112288A80000810000086DD600000000000FFFF00110000000000000000000000000000022002200000000000000000000000000022

PS_MODIFIERCOUNT [1] 1

PS_MODIFIER [1,0] 46 0xFFFF0000 RANDOM 1

PS_MODIFIERRANGE [1,0] 0 1 65535

PS_PACKETLENGTH [1] FIXED 88 1518

PS_PAYLOAD [1] INCREMENTING 0x00

PS_TPLDID [1] 1

PS_INSERTFCS [1] ON

PS_IPV4GATEWAY [1] 0.0.0.0

PS_IPV6GATEWAY [1] 0x00000000000000000000000000000000

PM_INDICES

PL_INDICES

PF_INDICES

PC_TRIGGER ON 0 FULL 0

PC_KEEP ALL 0 -1

PD_INDICES