OPENTEXT™

March 2017

Exceed TurboX

REST API User Guide, Version 11.5

This guide provides information and examples for accessing Exceed TurboX Server's REST APIs.

This guide is intended for software developers who wish to write scripts and programs to integrate with ETX server.

Support for REST APIs is provided by OpenText Professional Services.

Contents

Revision History 3
Overview 4
Why Use REST APIs? 4
Browsing the APIs 4
Using the APIs 6
HTTP Request Structure 6
HTTP Request URL 6
HTTP Request Methods 7
HTTP Request Headers 8
HTTP Request Body 8
HTTP Response Structure 9
HTTP Response Codes 9
HTTP Response Headers 10
HTTP Response Body 10
Need Help with APIs? 11
Appendix A: Accessing REST APIs with cURL 12
cURL Syntax 12
cURL Example: API Key Authentication 12
cURL Example: BASIC Authentication 12
cURL Example: POST (JSON from command line) 13
cURL Example: PUT (JSON from file) 14
cURL Example: Fetch Server Information 14
cURL Example: Check Node Utilization 15
cURL Example: Registering a Node 16
Appendix B: REST API Permissions 16
Appendix C: Advanced cURL Examples 19
Advanced cURL Example: Launch a Profile 19
Advanced cURL Example: Assign User to a Group 22
Advanced cURL Example: Modify a Profile 25
Appendix D: Node.js Example 28

Revision History

Revision Number	Date	Section modified	Modifications
1.1	March 2017	Appendix C	“Launch a Profile” example added
1.0	Oct 2016	All	Initial release

Overview

ETX server comes with a set of REST APIs that provide many benefits to users and server administrators. REST APIs are beneficial in a variety of situations, including:

- Speeding up administrative tasks
- Accessing ETX functions from a shell script or command line
- Accessing ETX from 3rd party websites or desktop applications
- Performing scheduled server maintenance

Why Use REST APIs?

REST APIs enable administrators and users to interact with the ETX Server via the HTTP/HTTPS protocol.

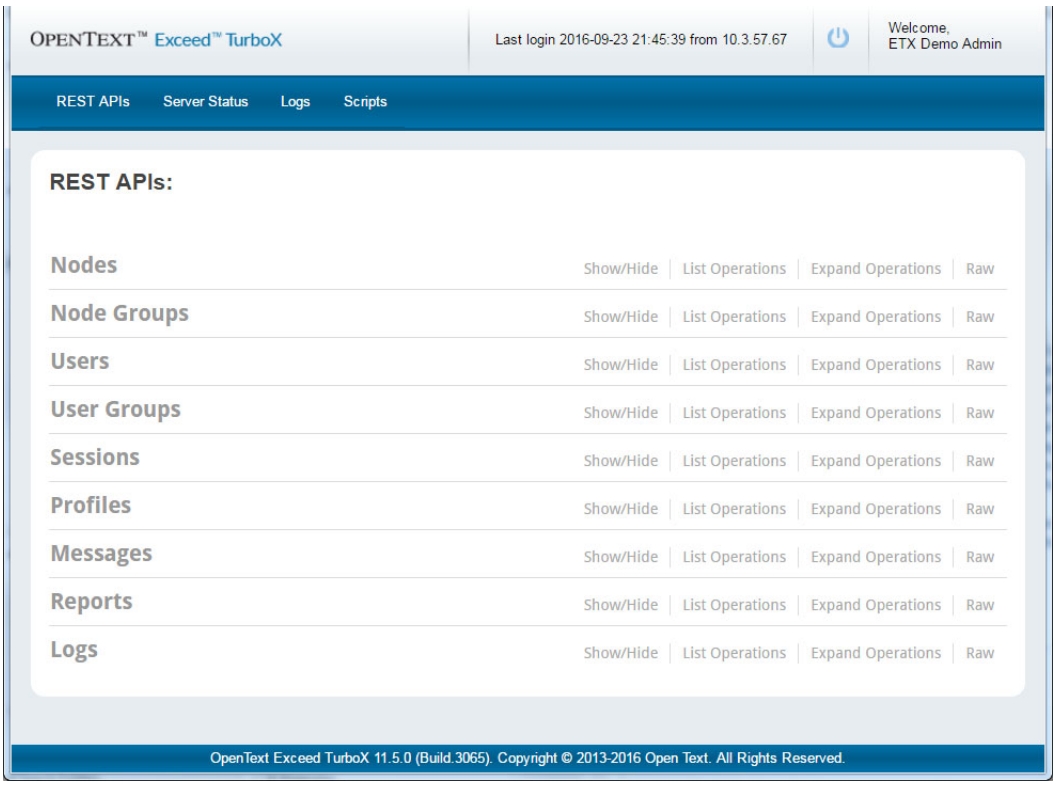
Using the APIs, you can perform the same functions as you would normally perform by accessing the ETX Server web interface, such as launching sessions, editing profiles, downloading reports, sending messages to users, creating node groups, and more.

Browsing the APIs

To see the available APIs, open your web browser and navigate to your ETX server login page. You will need to change the last part of the server URL to */etx/api*. For example:

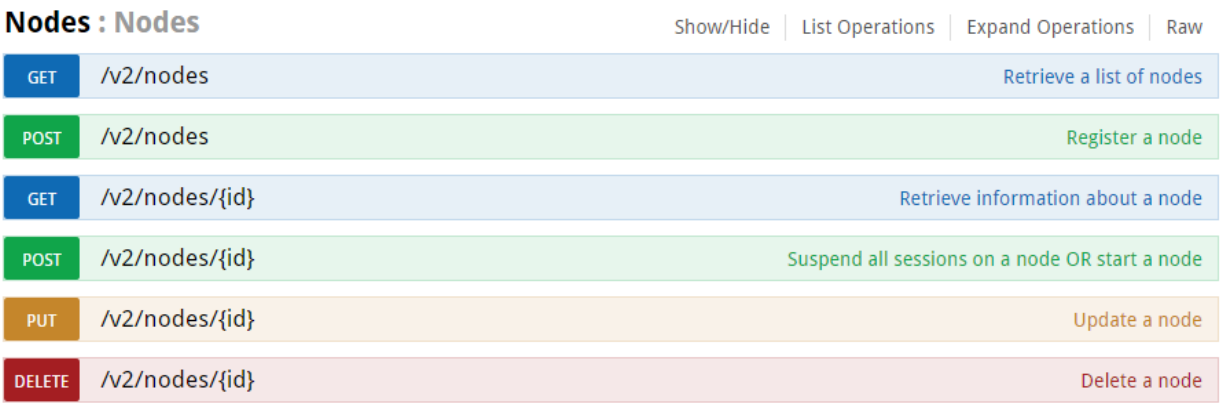
```
http://etx.xyz.com/etx/api
```

Once you log in with your ETX username and password, you can browse the list of available APIs:

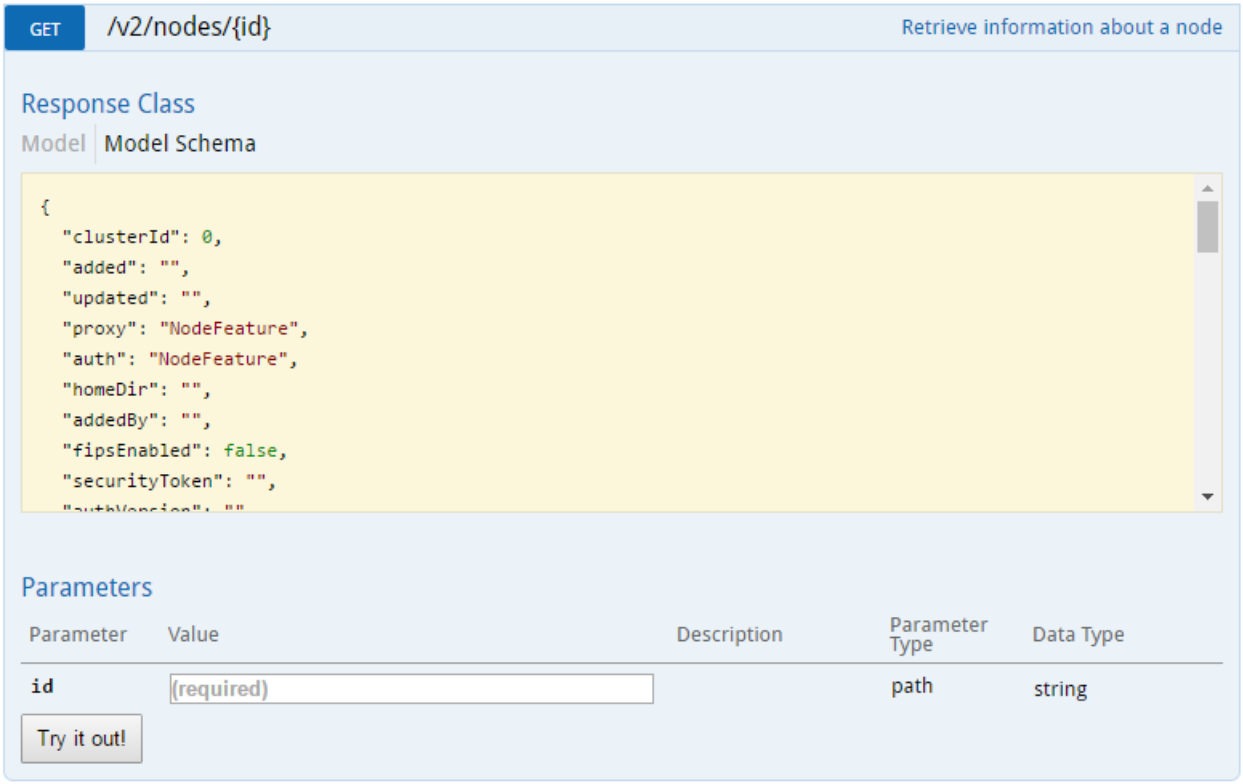


APIs are grouped into sections, based on the object you are accessing.

To see the actions you can perform on ETX nodes, click on the 'Nodes' header or 'Expand Operations' to see the list of available node APIs:



You can expand each of these APIs still further by clicking on them:



In this example, we are looking GET /v2/nodes/{id}, which is used to get information about an ETX connection node.

The built-in REST API documentation includes the following sections:

- Response Class (shows information about the object structure)
- Parameters (used for passing information to the API)

- "Try it out!" button (can be used to test the APIs prior to writing a script)

The self-documented APIs allow you to interact with the APIs from your browser. When learning how to use the APIs, you can use the "Try it out!" button to see the resulting HTTP request URL format and sample response codes.

Note: Some APIs may be hidden from you, depending on your account permissions. For example, regular ETX Users cannot access APIs relating to administrative functions. For a complete list of REST API permissions by user role, see Appendix B: REST API Permissions.

Using the APIs

In addition to viewing and interacting with the APIs in your browser, there are several other ways that you can use REST APIs:

- Command line utilities (e.g. cURL)
- Web browser extensions (e.g. Postman)
- Web programs (e.g. jQuery, node.js, PHP, Java)
- Native programs (e.g. C++)

Note: ETX Server REST APIs may be accessed with both http:// and https:// URLs. To use HTTP over SSL, simply change 'http' to 'https' (and include the HTTPS port number if not using port 443). For example:

```
https://etx.xyz.com/etx/api
```

For an example of how to access REST APIs from a UNIX/Linux or Windows command shell, see Appendix A: Calling REST APIs with cURL.

HTTP Request Structure

HTTP is the protocol that underlies all REST API calls. It is important to understand the structure of HTTP in order to access the ETX REST APIs.

An HTTP(S) request includes several key pieces of information:

- HTTP Request URL
- HTTP Request Method
- HTTP Request Headers
- HTTP Request Body

HTTP Request URL

The *HTTP Request URL* and *HTTP Request Method* together form the basic syntax of a REST API call.

The HTTP Request URL specifies the address of the API. It typically contains the following information:

- Server URL (e.g. `http://etx/xyz.com/etx/api`)
- Relative API Path (e.g. `/v2/nodes/5`)
- URL Parameters (not always required)

URL Parameters

URL parameters enable you to pass additional information to ETX server, which may be required in order for ETX Server to respond to your request. URL parameters are formatted as *query strings* using the following syntax:

```
?property1=value1&property2=value2
```

URL Parameter Example (Action)

```
POST http://etx.xyz.com/etx/api/v2/nodes/5?action=start
```

In this case, a URL parameter is used to specify an **action** on an object (in this case, starting the node).

URL Parameter Example (Filtering)

```
GET
http://etx.xyz.com/etx/api/v2/users?where_role=AdminFullAccess&where_user_group_id=4
```

In this example, a pair of **filters** limit a result set. Filters are optional parameters that you use in conjunction with GET APIs to narrow down lists of objects. Filter names always start with **where_**.

To understand which parameters are available for an API, you can log in to the ETX Server Core API page and browse the 'Parameters' section at the bottom of the API description. For more information on accessing the ETX Server APIs, see Browsing the APIs.

HTTP Request Methods

ETX supports the following HTTP request methods (also known as **HTTP Verbs**):

Method	Explanation	Examples
GET	Retrieve a collection of objects	GET /v2/users GET /v2/log/server/logfiles
GET	Retrieve an object by ID	GET /v2/sessions/100 GET /v2/users/155
POST	Create a new object	POST /v2/nodes POST /v2/users
POST	Send a message	POST /v2/messages/node/13 POST /v2/messages/session/855
POST	Perform an action on an object	POST /v2/sessions/50?action=resume POST /v2/profiles/10?distribute_to_user=8
PUT	Modify an existing object	PUT /v2/profiles/50 PUT /v2/usergroups/4
DELETE	Delete one or more objects	DELETE /v2/profiles/80 DELETE /v2/messages

Note: For the sake of simplicity, the examples above show only the relative API paths, rather than the full Request URL. For example, `GET /v2/users` would be replaced with:

```
GET http://etx.xyz.com/etx/api/v2/users
```

HTTP Request Headers

HTTP **headers** contain a variety of settings used for REST communication. The only headers you need to modify for REST calls are:

- Authorization: Specifies the type of authentication for the API call (see HTTP Authentication)
- Content-Type: Specifies the format of the HTTP Request Body

HTTP "Authorization" Header

All REST API calls require authorization. Users must identify themselves in order to access functions, so that ETX Server can restrict access to APIs based on ETX user privileges. For information on which APIs are accessible based on user role, see Appendix B: REST API Permissions

ETX supports two modes of REST authentication:

- **Basic authentication** (username and password are sent in plaintext, or as a base64 string)
- **API key authentication** (an ETX user creates an API key, and uses it to authenticate API calls)

API key authentication may be considered more secure than basic authentication for two reasons:

- Malicious users cannot steal your credentials from a script file or application code. A stolen API key enables the thief to access ETX with your credentials, but he/she cannot access other systems.
- API keys can have a timed expiry and can also be restricted to a single use. This eliminates the chance of active credentials being stolen after the timeout period.

For information on creating and managing API keys, please consult the ETX documentation.

For further information on HTTP headers, Wikipedia has a comprehensive listing of common headers. You should generally not need to modify any headers beyond the ones listed in this section.

HTTP Request Body

The HTTP Request Body is where you put JSON content for PUT and POST API calls.

Note: The HTTP Request Body is not required for GET and DELETE API calls, or some POST API calls which perform actions (which are specified as query string parameters in the URL) rather than creating objects.

JSON Content

JSON is a human-readable format for passing information about data objects. JSON is more compact than XML, making it a good format for passing information about large objects to REST APIs.

For examples of JSON formatting, see http://json.org/example.html

ETX uses JSON to pass and receive object data. GET APIs return objects in JSON format, while POST and PUT APIs accept JSON-formatted objects when creating or updating server resources.

When constructing a PUT or POST API call, you need to pass properly formatted JSON in the HTTP Request Body. The best way to do this is as follows:

1. Call the appropriate GET API to fetch a JSON-formatted object
2. Modify the JSON to include the values that you want
3. Pass the JSON in the Body of the HTTP request

For a real-world example of passing JSON-formatted text to an ETX Server API, see Accessing REST APIs with cURL.

HTTP Response Structure

ETX Server follows REST standards for HTTP responses. When an API has finished executing, the ETX Server returns an HTTP packet to the client. This HTTP response contains important information about the result of the API call.

The three sections of the HTTP response to pay attention to are:

- HTTP Response Code
- HTTP Response Headers
- HTTP Response Body

Each of these items is further explained in this section.

HTTP Response Codes

The following HTTP Response Codes are implemented by ETX Server. These response codes indicate the result of the API call.

HTTP Response Code	Meaning	Explanation
200	OK	Indicates successful completion of an API call. Check the HTTP Body for additional information, e.g. for the ID of an object created using POST.
204	No Content	Indicates successful completion of a DELETE request
400	Bad Request	The request could not be processed. Common causes are: • Incorrect object ID (object not found) • Missing or incorrect parameters • Missing or incorrect JSON content in Request Body
401	Unauthorized	Problem authenticating using the specified username / password / API key. Check your credentials / API key and try again.
403	Forbidden	Access to the API is not allowed for this user. For API permissions by user, see Appendix B: Rest API Permissions
404	Not Found	Incorrect URL

200-series response codes indicate a successful API call. 400 series codes indicate a problem with the request. When writing a script or program that calls REST APIs, it is a good idea to analyze the HTTP Response Code and react accordingly (e.g. by displaying the Response Code and Response Body to the user in the case of a 400-series error).

HTTP Response Headers

For the purposes of REST APIs, HTTP Response Headers can generally be ignored.

HTTP Response Body

The body of the HTTP response contains important information about the result of the API call.

The below table summarizes what HTTP Response Body is returned as a result of calling various REST APIs:

HTTP Verb	HTTP Response Code	HTTP Response Body	Explanation
GET	2xx (Success)	JSON object(s)	May return a single object or an object collection. Collections may be empty "[]".
GET	4xx (Failure)	Empty OR error message	
POST	2xx (Success)	JSON object(s)	JSON of the newly created object
POST	4xx (Failure)	Empty OR error message	
PUT	2xx (Success)	JSON object(s)	JSON of the newly updated object
PUT	4xx (Failure)	Empty OR error message	
DELETE	2xx (Success)	Empty	
DELETE	4xx (Failure)	Empty OR error message	

Need Help with APIs?

Support for REST APIs is provided by **OpenText Professional Services**.

Our team of consultants is exclusively comprised of OpenText employees with several years of experience in implementing and using our solutions. Our proven methodology, well-defined engagement rules and experience with thousands of customers are the guarantee that we will provide you with the highest possible quality of service to make your project a success.

Our Professional Services team can help you with all aspects of implementing and debugging Exceed TurboX Server's REST APIs, so that you can accomplish a variety of tasks, such as:

- **User interface development** of a new website or desktop application to launch applications and desktops using ETX REST APIs
- **Integration** of ETX into a corporate intranet or other web portal
- **Re-branding** of the web front-end to showcase your corporate logos, colors, and designs
- **Support and training** for developing your own custom scripts and libraries

For more information about our services, please contact your account representative or email us at: connsales@opentext.com.

Appendix A: Accessing REST APIs with cURL

cURL is a command-line tool that enables you to call REST APIs from a UNIX/Linux or Windows command shell. This allows you to do all sorts of useful things, like:

- Write scripts to accelerate common administrative tasks
- Access ETX Server functions from the command line
- Perform maintenance on ETX Server using a scheduled script (e.g. cron)

cURL is available on a wide variety of platforms and distributions, and at time of writing, is free for commercial use. You can download cURL here: https://curl.haxx.se/download.html

cURL Syntax

cURL HTTP requests use the following basic syntax:

```
curl [options...] <URL>
```

To see a full list of available options for curl, type 'curl --help' from the command line.

cURL Example: API Key Authentication

```
curl -H "Authorization: ETX omMusH9sTeG7Om0dkupoLQ" -k
https://etx.xyz.com/etx/api/v2/users
```

Command Snippet	Explanation
Curl	The curl utility
–k, -H	cURL options: -k: Allow SSL connection without a certificate -H: HTTP header. Must be followed by a string in the format "Property: Value". –H may be specified multiple times, once per header. To see a full list of options for curl, type 'curl --help' from the command line.
Authorization: ETX	Use API key authentication
omMusH9sTeG7Om0dkupoLQ	The API key

cURL Example: BASIC Authentication

Method #1:

```
curl -H "Authorization: BASIC dXNlcm5hbWU6cGFzc3dvcmQ=" -k
https://etx.xyz.com/etx/api/v2/users
```

Method #2:

```
curl--user username:password -k https://etx.xyz.com/etx/api/v2/users
```

Command Snippet	Explanation
Authorization: BASIC	Authorize using BASIC authentication (username and password)
dXNlcm5hbWU6cGFzc3dvcmQ=	Base 64 encoded username and password
--user username:password	Username and password

Note: You can use both of these methods for BASIC authentication. cURL internally converts method #2 into Base64 sets the Authorization header, so both of these commands look the same in the end. However, if you are not using cURL, you may need to manually convert your username and password to base64. In that case, you can use the following tool:

https://www.base64encode.org/

1. Click 'Encode' at the top
2. Paste your username and password into the top box, using the format *user:password*
3. Click the 'Encode' button and copy the result

cURL Example: POST (JSON from command line)

```
curl -v --user user1:passw0rd -X POST -H "Content-Type:application/json; charset=UTF-
8" --data "{\"name\":\"new profile
1\",\"iconURI\":\"\",\"userId\":\"40\",\"groupId\":\"0\",\"attrCollectionId\":\"0\",\"
attributes\":[{\"name\":\"EnableLog\",\"value\":\"1\"},{\"name\":\"LogFont\",\"value\"
:\"1\"}],\"xstarts\":[{\"name\":\"xterm\",\"startOrder\":0,\"locked\":false,\"attribut
eList\":[{\"name\":\"ApplicationNode\",\"value\":\"1\"},{\"name\":\"Command\",\"value\
":\"xterm\"}],\"startMethod\":0,\"id\":\"0\"}],\"type\":0,\"templateId\":\"1\"}"
http://etx.xyz.com/etx/api/v2/profiles
```

Command Snippet	Explanation
-v	cURL option (verbose)
-X POST	HTTP Request Method (POST)
-H "Content-Type: …"	cURL option (header) This sets the 'Content-Type' header, telling ETX Server that the HTTP Request Body is in JSON format with UTF-8 encoding.
--data "…"	cURL option (data) This option is immediately followed by a string that represents the HTTP Request Body in JSON format. Because cURL is a command line utility, quotes must be properly escaped (\") and the command must be on a single line. The following tool will convert multi-line JSON into a single line of JSON. You will need to change 'Indentation Level' to 'Compact': http://www.freeformatter.com/json-formatter.html#ad-output Once you have a single line of JSON, you can use this tool to escape the string for use on command line: http://www.freeformatter.com/javascript-escape.html#ad-output

cURL Example: PUT (JSON from file)

```
curl –X PUT -v --user admin:passw0rd -H "Content-Type:application/json" --data
@profile.txt  http://etx.xyz.com/etx/api/v2/profiles/5
```

Command Snippet	Explanation
-X PUT	HTTP Request Method (PUT)
-H "Content-Type:application/json"	cURL option (header) This sets the 'Content-Type' header, telling ETX Server that the file specified by -d is JSON format, with default encoding.
--data @profile.txt	cURL option (data) The -d flag supports passing a filename, which will insert the contents of a file in the HTTP Request Body.

cURL Example: Fetch Server Information

We want to check server status information using a shell command. In this example we want to check license usage to make sure we are not over our allotment:

In UNIX:

```
curl -s --user admin:passw0rd http://etx.xyz.com/etx/api/v2/log/server/stats | tr  ,
'\n' | grep License
```

In Windows PowerShell:

```
curl -s --user admin:passw0rd http://etx.xyz.com/etx/api/v2/log/server/stats | %
{$_.replace(",","`n")} | findstr -i License
```

This will print the following text:

```
"LicenseTransport":{"mode":[["OFF"]]}
"LicensesTotal":140
"LicensesInUse":200
```

This tells us that 140 of 200 licenses are in use.

Command Snippet	Explanation
-s	cURL option (header) -s is for silent mode, which tells cURL to only output the HTTP response body

| tr , '\n' | % {$_.replace(",","`n")}	Replaces commas in the JSON response with newline characters, which outputs each property on a separate line, making them easier to search
| grep License | findstr –i License	Searches the result for lines that contain the word 'License' and displays them to the console

cURL Example: Check Node Utilization

In this example we will check utilization of all ETX nodes from command line.

```
curl -s --user admin:passw0rd http://etx.xyz.com/etx/api/v2/nodes | tr  , '\n' | grep
'hostname\|cpu\|mem'
```

In Windows PowerShell:

```
curl -s --user admin:passw0rd http://etx.xyz.com/etx/api/v2/nodes | %
{$_.replace(",","`n")} | findstr -i "hostname cpu mem"
```

This prints the following text:

```
[{"hostname":"node1.xyz.com"
"cpu":2
"mem":35
{"hostname":"node2.xyz.com"
"cpu":1
"mem":77
{"hostname":"node3.xyz.com"
"cpu":0
"mem":0
{"hostname":"node4.xyz.com"
"cpu":11
"mem":6
{"hostname":"node5.xyz.com"
"cpu":0
"mem":0
```

Command Snippet	Explanation
| grep 'hostname|cpu|mem' | findstr –i "hostname cpu mem"	Searches for lines matching 'hostname' OR 'cpu' OR 'mem', and displays them in the order they appear.

If you have dozens of nodes, you may want to view the result one page at a time. In this case, add > file.txt at the end of the command, then call 'more file.txt' in DOS or command shell to view the file one page at a time.

cURL Example: Registering a Node

Node registration is performed at the end of the node installation. If for some reason this registration didn't happen, or if you are manually copying the files and registering the node, ETX provides a 'nodecmds' utility for registering the node. This utility is in the /bin folder at the root of the node installation directory. For example:

```
./bin/nodecmds registernode
for interactive mode, or:
./bin/nodecmds maint registernode https://etx.xyz.com:443 -keyauth 5510 proxy:1
appscan:0
for non-interactive mode (will prompt for API Key)
```

ETX also provides a REST API for registering a node. Installation of a node via REST requires the following steps:

1. Create a text file (e.g. "node.json") containing basic node settings in JSON format:

```
{
  "port": 5510,
  "hostname" : "testnode1.com",
  "proxy" : "Enabled",
  "auth" : "Disabled"
}
```

2. Register the node using cURL:

```
curl --user admin:pwd -X PUT -H "Content-Type: application/json" --data @node.json
http://etx.xyz.com/etx/api/v2/nodes?action=start
```

This command not only starts the node, but also creates it from the JSON in a single action.

Appendix B: REST API Permissions

For security reasons, each REST API is only visible and available to users with the appropriate ETX user permissions. You can use the following table to determine if a user has access to any given REST API.

Legend

Y: API is fully available. API calls by this user affect all objects for all users.
N: API is unavailable. API calls by this user return HTTP 403 (Forbidden).
Y*: API is partially available. APIs calls by this user affect only the user's objects.
Y**: Same as Y* but additional user permissions apply.

	Admin (Full)	Admin (Read-Only)	Session Manager	Technical Support	User
GET /v2/nodes	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/nodes/{id}	**Y**	**Y**	**Y**	**Y**	**N**
POST /v2/nodes/	**Y**	**N**	**N**	**N**	**N**
POST /v2/nodes/{id}?action=start	**Y**	**N**	**Y**	**Y**	**N**
POST /v2/nodes/{id}?action=suspend_all_sessions	**Y**	**N**	**Y**	**Y**	**N**
PUT /v2/nodes/{id}	**Y**	**N**	**Y**	**N**	**N**
DELETE /v2/nodes/{id}	**Y**	**N**	**N**	**N**	**N**
GET /v2/nodegroups/	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/nodegroups/{id}	**Y**	**Y**	**Y**	**Y**	**N**
POST /v2/nodegroups/	**Y**	**N**	**Y**	**N**	**N**
PUT /v2/nodegroups/{id}	**Y**	**N**	**Y**	**N**	**N**
DELETE /v2/nodegroups/{id}	**Y**	**N**	**Y**	**N**	**N**
GET /v2/users	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/users/{id}	**Y**	**Y**	**Y**	**Y**	**Y***
POST /v2/users/	**Y**	**N**	**N**	**N**	**N**
PUT /v2/users/{id}	**Y**	**Y***	**Y***	**Y***	**Y***
DELETE /v2/users/{id}	**Y**	**N**	**N**	**N**	**N**
GET /v2/usergroups	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/usergroups/{id}	**Y**	**Y**	**Y**	**Y**	**N**
POST /v2/usergroups/	**Y**	**N**	**N**	**Y**	**N**
PUT /v2/usergroups/{id}	**Y**	**N**	**N**	**Y**	**N**
DELETE /v2/usergroups/{id}	**Y**	**N**	**N**	**Y**	**N**
GET /v2/sessions	**Y**	**Y**	**Y**	**Y**	**Y***
GET /v2/sessions/{id}	**Y**	**Y**	**Y**	**Y**	**Y***
POST /v2/sessions/{id}?action=suspend	**Y**	**Y***	**Y**	**Y**	**Y***
POST /v2/sessions/{id}?action=resume	**Y**	**Y***	**Y**	**Y**	**Y****
POST /v2/sessions/launch/{profileId}	**Y***	**Y***	**Y***	**Y***	**Y***
DELETE /v2/sessions/{Id}	**Y**	**Y***	**Y**	**Y**	**Y***
GET /v2/profiles/	**Y**	**Y**	**Y**	**Y**	**Y***

GET /v2/profiles/{Id}	**Y**	**Y**	**Y**	**Y**	**Y****
POST /v2/profiles/	**Y**	**Y**	**Y**	**Y**	**Y****
POST /v2/profiles/{id}?distribute_to_user={userId}	**Y**	**N**	**N**	**Y**	**N**
POST /v2/profiles/{id}?distribute_to_usergroup={usergroupId}	**Y**	**N**	**N**	**Y**	**N**
PUT /v2/profiles/{id}	**Y**	**Y***	**Y***	**Y**	**Y***
DELETE /v2/profiles/{id}	**Y**	**Y***	**Y***	**Y**	**Y***
GET /v2/messages/	**Y***	**Y***	**Y***	**Y***	**Y***
POST /v2/messages/node/{nodeId}	**Y**	**N**	**Y**	**Y**	**N**
POST /v2/messages/nodegroup/{nodeGroupId}	**Y**	**N**	**Y**	**Y**	**N**
POST /v2/messages/user/{userId}	**Y**	**N**	**Y**	**Y**	**N**
POST /v2/messages/usergroup/{usergroupId}	**Y**	**N**	**Y**	**Y**	**N**
POST /v2/messages/session/{sessionId}	**Y**	**N**	**Y**	**Y**	**N**
DELETE /v2/messages/	**Y***	**Y***	**Y***	**Y***	**Y***
DELETE /v2/messages/{Id}	**Y***	**Y***	**Y***	**Y***	**Y***
GET /v2/reports/sessions	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/reports/licenses	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/log/server/activity/	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/log/server/logfiles	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/log/server/logfiles/{filename}	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/log/server/stats	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/log/node/crash-reports/{id}	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/log/node/crash-reports/{id}/{filename}	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/log/node/logfiles/{id}	**Y**	**Y**	**Y**	**Y**	**N**
GET /v2/log/node/logfiles/{id}/{filename}	**Y**	**Y**	**Y**	**Y**	**N**

Appendix C: Advanced cURL Examples

This section includes a couple of shell scripts that perform more advanced functions using cURL from UNIX command line. These examples are provided as-is; OpenText does not guarantee that these examples will work in your environment.

Advanced cURL Example: Launch a Profile

ETX profiles can be launched using REST if the ETX native client launcher is installed (the Java launcher cannot be used to launch profiles via REST). The client launcher can be downloaded from the user settings panel on the ETX dashboard.

The following script accepts a server name, server HTTP port number, username, password, profileID and launches a profile using these parameters. You can modify the script to use an API key instead of username and password for authentication, or switch from HTTP to HTTPS by making small changes to the script if needed.

Example #1: Windows Shell

```
rem usage: launch.cmd server:test01.lab.opentext.com port:80 user:guest
pass:stormY!12 id:9

@echo off

set port=80
set server=myserver.company.com
set user=
set pass=
set id=9

:parseparameters
IF "%~1"=="" GOTO launch
set param=%~1
for /F "tokens=1,2 delims=:" %%a in ("%param%") do (

  if "%%a"=="server" set server=%%b
  if "%%a"=="port" set port=%%b
  if "%%a"=="user" set user=%%b
  if "%%a"=="pass" set pass=%%b
  if "%%a"=="id" set id=%%b
)
SHIFT
GOTO parseparameters

:launch
set curlCmd=curl -v -u %user%:%pass% -X POST -H "Content-
Type:application/json" http://%server%:%port%/etxcore/v2/sessions/launch/%id%
set jqCmd=jq-win64.exe -r ".uriEtxLauncher"
set pipeCmd="%curlCmd% | %jqCmd%"

FOR /F "tokens=* USEBACKQ" %%F IN (` %pipeCmd% `) DO (
SET uriEtxLauncher=%%F
)
echo %uriEtxLauncher%
#cmd /c
```

```
start "" "%uriEtxLauncher%"
```

Example 2: UNIX Shell

```
#!/bin/bash
# launch profile.
# Copyright (c) 2016 Open Text. All Rights Reserved.
#./launch.sh -u guest -pwd stormY!12 -s test01.lab.opentext.com -p 80 -id 8
#

port=80
home=$HOME

while [ "$1" != "" ]; do
    case $1 in
        -u | --user ) shift
            user=$1
            ;;
        -pwd | --password ) shift
            password=$1
            ;;
        -s | --server ) shift
            server=$1
            ;;
        -p | --port ) shift
            port=$1
            ;;
        -id | --id ) shift
            id=$1
            ;;
    esac
    shift
done

if [[ -z "${user}" || -z "${password}" || -z "${server}" || -z "${id}" ]];
then
cat <<help
-u,--user: username
-pwd,--password: password
-s,--server: server
-p,--port: port,default 80
-id,--id: profile id
sample:./launch.sh -u etxadmin -pwd exceeddemo -s connrh6501.lab.opentext.com
-p 80 -id 9

help
exit 1
fi

userpass=${user}:${password}
uriEtxLauncher=`curl -v -u ${userpass} -X POST -H "Content-
Type:application/json"
http://${server}:${port}/etxcore/v2/sessions/launch/${id} | jq -r
'.uriEtxLauncher'`

echo $uriEtxLauncher
```

```
#curl -v -u $uriEtxLauncher
xdg-open $uriEtxLauncher
```

Example 3: HTML

This example requires jquery-1.11.1.min.js to be saved to the same folder as the HTML page. You can get jquery here: https://code.jquery.com/jquery-1.11.1.min.js

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta http-equiv="X-UA-Compatible" content="IE=Edge">
    <meta http-equiv="Cache-Control" content="no-cache">
    <meta http-equiv="Pragma" content="no-cache">
    <meta http-equiv="expires" content="-1">
    <script src="jquery-1.11.1.min.js"></script>

    <script type="text/javascript">
        $(document).ready(function () {
            $('#launch').click(function () {
                var username = $('#username').val();
                var password = $('#password').val();
                var server = $('#server').val();
                var port = $('#port').val();
                var profileId = $('#profileId').val();
                var launchInfoUrl =
"http://"+server+":"+port+"/etxcore/v2/sessions/launch/"+profileId;

                $.ajax({
                        type: "POST",
                        url: launchInfoUrl,
                        contentType: "application/json; charset=utf-8",
                        crossDomain: true,
                        dataType: "json",
                        headers: {
                            "Authorization": "Basic " + btoa(username + ":" +
password)
                        },
                        success: function (data, status, jqXHR) {
                            var launcherUri = data.uriEtxLauncher;
                            window.location.href = launcherUri;
                        },

                        error: function (jqXHR, status) {
                        }
                    });
            });

        });
    </script>
</head>
<body>
<table style="width:100%">
```

```
  <tr>
    <td>username</td>
    <td><input type="text" id="username" value=""></td>
  </tr>
  <tr>
    <td>password</td>
    <td><input type="text" id="password" value=""></td>
  </tr>
  <tr>
    <td>server</td>
    <td><input type="text" id="server" value="wangvcentos01"></td>
  </tr>
  <tr>
    <td>port</td>
    <td><input type="text" id="port" value="8080"></td>
  </tr>
  <tr>
    <td>profile id</td>
    <td><input type="text" id="profileId" value="9"></td>
  </tr>
  <tr>
    <td colspan=2>for cross domain rest api call, please config CORS filter
in etxcore/web-inf/web.xml </td>
  </tr>
  <tr>
    <td></td>
    <td><input type="button" id="launch" value="launch"></td>
  </tr>
</table>

</body>
</html>
```

Advanced cURL Example: Assign User to a Group

It is often the case that a new user comes on board and LDAP groups are not conveniently mapped to allow the new user automatic access to common group profiles. In this case it is necessary to add the new user to a custom ETX group.

The following script accepts the server URL, login name for the user to add to a group, and the group name to add them to. It makes use of the jq tool (https://stedolan.github.io/jq) in order to parse the JSON returned by the server. The script checks to see if the user is included in the group or not before finally writing the group state back.

```
#!/bin/bash

function reset_env()
{
JQ_Loginname=
export JQ_Loginname
JQ_Groupname=
export JQ_Groupname
JQ_Groupid=
```

```
export JQ_Groupid
JQ_Userid=
export JQ_Userid
JQ_newmembers=
export JQ_newmembers
}

SERVERURL=$1
Loginname=$2
Groupname=$3

JQ=`which jq`
if [ -z "$JQ" ] ; then
   echo "This script relies on jq from https://stedolan.github.io/jq"
   echo "You can either modify the script to run from specified path"
   echo "or add jq to path"
   exit 1
   # you can modify this script here - comment out the exit and change line
below
   JQ=/opt/yourjqpath
fi

if [ "$4" = "-t" ] ; then
    RESTApiKey=$5
    CURLAUTH=" -H \"Authorization: ETX ${RESTApiKey}\" "
else
    RESTUser=$4
    RESTPasswd=$5

    if [ -z "${RESTUser}" ] ; then
      echo "Please enter the user account to perform this operation"
      read RESTUser
    fi
    if [ -z "${RESTPasswd}" ] ; then
      echo "Please enter the password for '${RestUser}'"
                        stty -echo
      read RESTPasswd
                        stty echo
    fi

    CURLAUTH=" -u ${RESTUser}:${RESTPasswd} "
fi

if [ -z "$SERVERURL" -o -z "$Loginname" -o -z "$Groupname" ] ; then
   echo "invalid arguments"
   echo "SERVERURL  loginname  groupname {auth_strings}"
   echo "where "
   echo "  {auth_strings} can be either:"
   echo "   -t apikey"
   echo "   username [password]"
   echo "         blank password will cause prompt"
   exit 1
fi

#  Easiest way for jq to use env variables is to export the values
JQ_Loginname=$Loginname
export JQ_Loginname
```

```
JQ_Groupname=$Groupname
export JQ_Groupname

# get group that matches provided name for group
GROUPID=`curl -s -GET $CURLAUTH$SERVERURL/etx/api/v2/usergroups | $JQ '.[] |
select(.name==env.JQ_Groupname) |.id' | sed -e 's/^"//' -e 's/"$//'`
if [ -z "$GROUPID" ] ; then
   echo "Error - could not retrieve GroupId for $Groupname"
   reset_env
   exit 1
fi

# get userid for user using similar method
USERID=`curl -s -GET $CURLAUTH$SERVERURL/etx/api/v2/users | $JQ '.[] |
select(.login==env.JQ_Loginname) |.id' | sed -e 's/^"//' -e 's/"$//'`
if [ -z "$GROUPID" ] ; then
   echo "Error - could not retrieve User Id for $Loginname."
   reset_env
   exit 1
fi

JQ_Groupid=$GROUPID
export JQ_Groupid
JQ_Userid=$USERID
export JQ_Userid


# assign this user into this group
curl -s -GET $CURLAUTH$SERVERURL/etx/api/v2/usergroups/$GROUPID -o
tmp_thisgrp.json
members=`$JQ -e '.members' tmp_thisgrp.json| sed -e 's/^"//' -e 's/"$//'`
if [ "${members}" = "0" ] ; then
    # easiest - no members so just add one
    $JQ -e '.members=env.JQ_Userid' tmp_thisgrp.json > updategrp.json
else
    # check if the userid is included in the list
    origmembers=$members
    while [ "$members" ] ;do
      iter=${members%%,*}
      if [ "$iter" = "$USERID" ] ; then
          echo "User $Loginname is already included in group $Groupname"
          reset_env
          exit 0
      fi
      [ "$members" = "$iter" ] &&  members='' ||  members="${members#*,}"
    done
    JQ_newmembers="${origmembers},$USERID"
    export JQ_newmembers
    # update the json with the new member in there
    $JQ -e '.members=env.JQ_newmembers' tmp_thisgrp.json > updategrp.json
fi

curl -s -X PUT -H "Content-Type: application/json; charset=UTF-8"  -d
@updategrp.json $CURLAUTH$SERVERURL/etx/api/v2/usergroups/$GROUPID
result=$?

rm -f tmp_thisgrp.json
```

```
rm -f updategrp.json
reset_env
exit $result
```

Advanced cURL Example: Modify a Profile

The following example uses JavaScript to GET a profile, modify the JSON 'TargetNode' property, and PUT the profile back to the server. In addition to JavaScript, REST APIs can be accessed from most popular web and desktop programming languages such as PHP, Java, C, and Ruby.

How to run this shell script (from Linux console):

1. Download and install cURL
2. Copy the shell script below into a file named 'updateprofile.sh'
3. Call the shell script via command line, passing all necessary parameters

Example usage:

```
./updateprofile.sh -u admin -pwd testing -s etx.xyz.com -p 8080 -id 18 -n1 TargetNode
-v1 "Single>newnode.xyz.com" -n2 EnableLog -v2 0 -n3 LogFont -v3 1
```

```
#!/bin/bash

port=80
i=0
j=0

# parse the command line parameters
while [ "$1" != "" ]; do
    case $1 in
        -u | --user ) shift
            user=$1
            ;;
        -pwd | --password ) shift
            password=$1
            ;;
        -s | --server ) shift
            server=$1
            ;;
        -p | --port ) shift
            port=$1
            ;;
        -id | --id ) shift
            id=$1
            ;;
        -n[1-999] | --name[1-999] ) shift
            names[${i}]=$1
            i=$(( $i + 1 ))
            ;;
        -v[1-999] | --value[1-999] ) shift
            values[${j}]=$1
```

```
            j=$(( $j + 1 ))
            ;;
    esac
    shift
done

# If syntax is incorrect, display a help prompt to the user
if [[ -z "${user}" || -z "${password}" || -z "${server}" || -z "${id}" ]];
then
cat <<help
-u,--user: username
-pwd,--password: password
-s,--server: server
-p,--port: port,default 80
-id,--id: profile id
-n1,--name1 : attribute name, replace variable hoder(#{n1},#{n2},#{n3}) in
json file
-v1,--value1 : attribute value, replace variable hoder(#{v1},#{v2},#{v3}) in
json file
help
exit 1
fi

# construct the authentication string
userpass=${user}:${password}

# output JSON to a temporary file
jsonFilePut=profile${id}put.json;
etxapilog=updateprofile.log;

# GET the profile into a JSON file
curl -u ${userpass} http://${server}:${port}/etx/api/v2/profiles/${id}|jq '.'
> ${jsonFilePut}

# iterate through the JSON using jq and update the JSON attributes with
values passed from command line
for (( k=0; k<$i; k++ ))
do
echo ${names[$k]}
cat ${jsonFilePut} | jq --arg attrName "${names[$k]}" --arg attrValue
"${values[$k]}" 'def walk(f):
  . as $in
  | if type == "object" then
      reduce keys[] as $key
        ( {}; . + { ($key):  ($in[$key] | walk(f)) } ) | f
  elif type == "array" then map( walk(f) ) | f
  else f
  end;walk(if type == "array" then map(if .name == $attrName then .
+{"value":$attrValue} else . end) else . end)' >${jsonFilePut}
done

cat ${jsonFilePut} | tee -a $etxapilog

# PUT the modified JSON back to the server
curl -v -u ${userpass} -X PUT -H "Content-Type:application/json"
http://${server}:${port}/etx/api/v2/profiles/${id} -d @${jsonFilePut} 2>&1|
tee -a $etxapilog
```

```
# delete the temporary file
rm ${jsonFilePut}
```

Appendix D: Node.js Example

The following example uses JavaScript to GET a profile, modify the JSON 'TargetNode' property, and PUT the profile back to the server. In addition to JavaScript, REST APIs can be accessed from most popular web and desktop programming languages such as PHP, Java, C, and Ruby.

How to run this code (from Linux console):

1. Download and install node.js
2. Save the source code below to a file named 'updateprofile.js'
3. This example is designed to accept parameters from command line interface. So you will need to specify the server name, user login, profile ID, and profile properties to change.

Example usage:

```
node updateprofile.js user=admin password=testing server=etx.xyz.com port=8080 id=18
TargetNode="Single>newnode.xyz.com" EnableLog=1 LogFont=0
```

```
// import https and filesystem libraries
var http = require('http');
var fs = require('fs');

// initialize a log file
var logger = fs.createWriteStream('updateprofile.log');
process.stdout.write = process.stderr.write = logger.write.bind(logger);

// read command params
if (process.argv.length <= 2) {
    process.exit(-1);
}
var args = process.argv.slice(2);

var server;
var port;
var username;
var password;
var id;
var attributes = [];

// parse command params
for (var i = 0; i < args.length; i++) {
    var param = args[i].split("=");
    if (param[0] == "server") {
        server = param[1];
    } else if (param[0] == "port") {
        port = param[1];
    } else if (param[0] == "user") {
        username = param[1];
    } else if (param[0] == "password") {
        password = param[1];
    } else if (param[0] == "id") {
        id = param[1];
    } else {
        attributes[param[0]] = param[1];
```

```
    }
}

// Contruct http request objects for GET and PUT calls
var authToken = new Buffer(username + ":" + password).toString('base64')
var optionsGet = {
    host: server,
    port: port,
    path: '/etx/api/v2/profiles/' + id,
    method: 'GET',
    headers: {
        'Authorization': 'BASIC ' + authToken
    }
};
var optionsPut = {
    host: server,
    port: port,
    path: '/etx/api/v2/profiles/' + id,
    method: 'PUT',
    headers: {
        'Authorization': 'BASIC ' + authToken,
        'Content-Type': 'application/json'
    }
};

// Call the GET API to get contents of the specified profileID
http.get(optionsGet, function (res) {
    console.log("Got response: " + res.statusCode);
    console.log('HEADERS: ' + JSON.stringify(res.headers));
    var jsonData = '';
    res.on('data', function (chunk) {
        jsonData += chunk;

    });
    res.on('end', function () {
        if (res.statusCode != "200") {
            console.log("error:no permssion or bad request!");
            return;
        }
        updateAttributes(jsonData, attributes);
    });

}).on('error', function (e) {
    console.log("Got error: " + e.message);
});

// PUT the object back with updated attributes
function updateAttributes(jsonData, modifedAttributes) {

    // Take the profile JSON body from the GET, and set the JSON properties
to the values passed from the command line
    var data = JSON.parse(jsonData);
    data.attributes.forEach(function (attrObj, index) {
        var newValue = modifedAttributes[attrObj.name];
        if (newValue != null) {
            attrObj.value = newValue;
        }
```

```
    });

    // Construct the HTTP PUT request
    var req = http.request(optionsPut, function (res) {
        console.log('STATUS: ' + res.statusCode);
        console.log('HEADERS: ' + JSON.stringify(res.headers));
    });

    req.on('error', function (e) {
        console.log("Got error: " + e.message);
    });

    var jsonDataPut = JSON.stringify(data);
    console.log("json data:");
    console.log(data);

    // PUT the data via HTTP
    req.write(jsonDataPut);
    req.end();

}
```