

IxLoad

REST API Programming Guide

Version 8.40EA-Update1

January 2018



Notices

Copyright Notice

© Keysight Technologies 2015–2018

No part of this document may be reproduced in any form or by any means (including electronic storage and retrieval or translation into a foreign language) without prior agreement and written consent from Keysight Technologies, Inc. as governed by United States and international copyright laws.

Warranty

The material contained in this document is provided “as is,” and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Keysight disclaims all warranties, either express or implied, with regard to this manual and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Keysight shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Keysight and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

U.S. Government Rights

The Software is "commercial computer software," as defined by Federal Acquisition Regulation ("FAR") 2.101. Pursuant to FAR 12.212 and 27.405-3 and Department of Defense FAR Supplement ("DFARS") 227.7202, the U.S. government acquires commercial computer software under the same terms by which the software is customarily provided to the public. Accordingly, Keysight provides the Software to U.S. government customers under its standard commercial license, which is embodied in its End User License Agreement (EULA), a copy of which can be found at <http://www.keysight.com/find/sweula> or <https://support.ixiacom.com/support-services/warranty-license-agreements>. The license set forth in the EULA represents the exclusive authority by which the U.S. government may use, modify, distribute, or disclose the Software. The EULA and the license set forth therein, does not require or permit, among other things, that Keysight: (1) Furnish technical information related to commercial computer software or commercial computer software documentation that is not customarily provided to the public; or (2) Relinquish to, or otherwise provide, the government rights in excess of these rights customarily provided to the public to use, modify, reproduce, release, perform, display, or disclose commercial computer software or commercial computer software documentation. No additional government requirements beyond those set forth in the EULA shall apply, except to the extent that those terms, rights, or licenses are explicitly required from all providers of commercial computer software pursuant to the FAR and the DFARS and are set forth specifically in writing elsewhere in the EULA. Key-sight shall be under no obligation to update, revise or otherwise modify the Software. With respect to any technical data as defined by FAR 2.101, pursuant to FAR 12.211 and 27.404.2 and DFARS 227.7102, the U.S. government acquires no greater than Limited Rights as defined in FAR 27.401 or DFAR 227.7103-5 (c), as applicable in any technical data. 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Table of Contents

New in this Release	1
Before you Begin	2
REST Resources	2
Supported Features.....	3
Using the REST API over HTTPS.....	3
Self-signed certificates	3
Script Changes Required for HTTPS	4
Errors from REST UI Clients.....	5
REST Authentication.....	5
Prerequisites	5
Enabling Authentication	5
Ixia User Management.....	6
Authenticating REST Requests	6
Retrieving the api-key	7
Script Changes Required for Authentication	7
Supported methods and running operations.....	8
REST representation	8
Primitive values.....	8
List Objects.....	8
REST resources.....	8
Case conventions.....	9
Preferences	9
IxLoad REST Methods.....	9
GET.....	9
PATCH	11
POST.....	11
DELETE	12
OPTIONS	13
Operations	15
Starting an operation.....	15
uploadFile	16
Getting an Operation's Status.....	16
Examples of Common Operations in the IxLoad REST API.....	18
Query Strings	21

Collecting Diagnostics	21
IxLoadGateway - IxLoad session handling.....	23
Creating a new session	23
Starting a session	24
Deleting a session	27
IxLoad Data Model	27
Communities	27
Timelines	28
Login name.....	28
Modifying the Activity user Objective Value on the fly	28
Chassis Chain / Port Assignment Operations	29
Adding a chassis	29
Connecting to a chassis.....	30
Removing a chassis	32
Unassign ports	32
Assign ports.....	32
IxVM Chassis (ixChassisBuilder)	33
Statistics	36
Viewing statistics	36
RunState Stat Source	38
Modifying configured statistics.....	38
Filtering stats	40
Adding an Activity Filter.....	43
Generated CSVs	43
Logging	43
REST Script Templates.....	44
AddNewCommand.....	45
ChangeAgentObjectives.....	45
ChangeIpType	45
SimpleRun	46
IxLoadRestUtils.....	46
class Connection(__builtin__.object)	46
class WebList(__builtin__.list)	47
class WebObject(__builtin__.object).....	48
Functions	48
IxLoadUtils.....	48
Functions	48

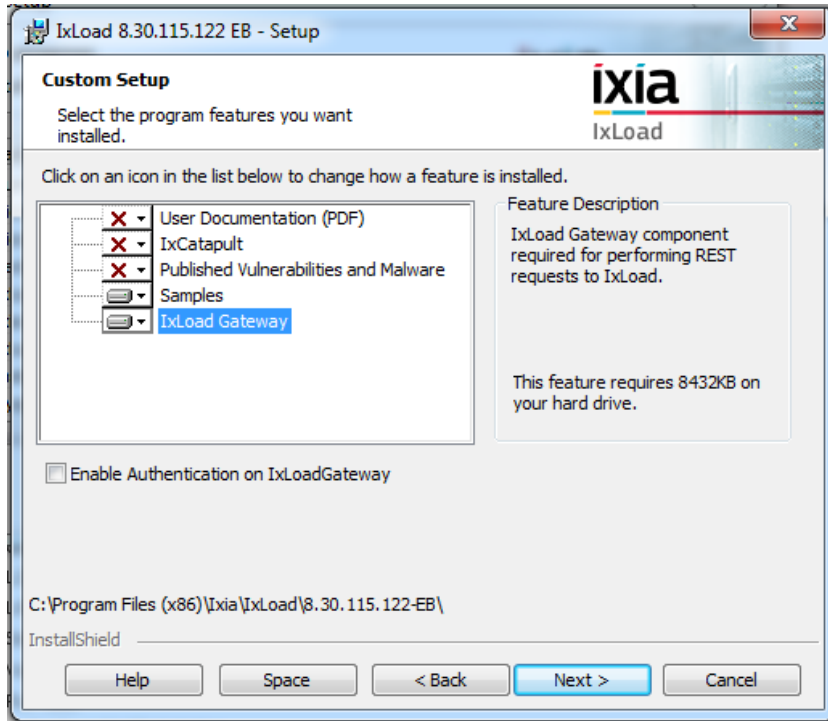
New in this Release

The following features are new in this release:

HTTP to HTTPS redirection	The REST API now redirects HTTP requests to HTTPS transport. See <i>Using the REST API over HTTPS</i> for details.
crf import	You can import crf files through the REST API. See <i>Example for Importing a .crf File</i> .
waitForAllCaptureData	A request that waits for all the port data to be captured has been added. See <i>Example for Waiting to Capture the Port Data</i> .
Modifying the activity user objective on the fly	A request that changes the activity user objective while the test is running has been added. See <i>Modifying the Activity user Objective Value on the fly</i> .
Activity filters	A request that adds an activity filter to a statistic has been added. See <i>Adding an Activity Filter</i> .

Before you Begin

The IxLoad REST API requires the IxLoad Gateway Service to be installed on the PC where you will use the REST API. The Gateway Service is an optional component that is not installed by default. To install it, you must select Custom Setup during IxLoad installation. If you have already installed IxLoad, you can select IxLoad from the list of installed applications (Control Panel | Programs and Features), right-click and select Modify. The installer will run and you can install the IxLoad Gateway service.



REST Resources

A resource is a basic concept in terms of Rest. This chapter will define the resource definition in terms of IxLoad.

A resource is a representation of an IxLoad object for the user. Not all IxLoad objects will be resources and not all IxLoad object functionality will be exposed to the user.

A resource can have:

- Properties:
 - Primitives: simple types like: bool , int ,string
 - Complex: other resources: timeline resources, agent resources and so on.
- Operations:
 - Example of operations
 - RunTest
 - AddChassis
 - RefreshChassis

The IxLoad REST API enables starting and configuring an IxLoad session through REST API, via HTTP requests.

Supported Features

Below is a list of features that is supported in this release of the IxLoad REST API:

- Create and start an IxLoad session
- Load a configuration (.rx). The rx will be loaded from a local path.
- View data model tree, via GET requests (including query string support)
- Remove existing chassis and add new chassis
- Assign and un-assign ports
- Change existing configuration - modify field values via PATCH requests
- L23 range support
- L47 plugin support
- Save configuration modified through REST API
- View / add / delete configured L23 and L47 statistics.
- Run test
- Poll L23 and L47 statistics
- Upload repository (.rx) files
- Start remote IxLoad sessions
- Automatically generated documentation
- Querying logs from REST API
- Analyzer

The following list of features is not supported in this release of the IxLoad REST API:

- Applibrary protocols/ Resource Manager / Profiles (e.g. Real Files)
- IxReporter
- Adding (POST) / removing (DELETE) objects such as test communities, plugins, ranges. Add and remove operations are only supported officially on the chassis list, the port lists and configured stats.

Using the REST API over HTTPS

Starting with IxLoad 8.40 release, requests made through IxLoad REST API are supported over both HTTP and HTTPS transport. The HTTP requests will be redirected by IxLoadGateway to the HTTPS server and translated into HTTPS requests.

The default starting port for the IxLoadGateway HTTP server is 8080. Therefore, you can access IxLoadGateway through HTTP requests on a URL in the following format:

http://<IP_ADDRESS>:8080/api/v0/sessions

The default starting port for the IxLoadGateway HTTPS server is 8443. Therefore, you can access IxLoadGateway through HTTPS requests on a URL in the following format:

https://<IP_ADDRESS>:8443/api/v0/sessions

Self-signed certificates

HTTPS support over IxLoad REST API is offered through a self-signed certificate that is automatically generated by the IxLoad Gateway component when it is installed as part of an IxLoad installation.

The self signed-certificate consists of two files:

- ixload_certificate.crt – this represents the actual self-signed certificate
- ixload_privkey.key – this represents the private key used by the self-signed certificate

Depending on the operating system on which the IxLoad build was installed, the self-signed certificate and its corresponding private key can be found at the following locations:

- On a Windows OS, they can be found at `<IxLoadGateway_Install_Path>\certificate`
- On the IxLoad Linux OVA, they can be found at `/opt/ixia/ixloadgateway/certificate`

The self-signed certificate is generated using a 2048 bit RSA keypair and the SHA-256 signature hash algorithm.

The self-signed certificate includes an X509 extension known as SNI (Subject Name Identifier)/SAN (Subject Alternative Name). This extension allows the certificate to specify under which names (hostnames, IP addresses, etc) a user can access a secured web server that is using that certificate. This will prevent users from accessing IxLoad Gateway instances on different machines using the same self-signed certificate.

For this extension, IxLoad Gateway generates a log file named `san.log` which contains all the hostnames and IPv4/IPv6 addresses under which the machine where IxLoad Gateway is installed can be accessed. This log file resides in the same location as the autogenerated certificate.

The certificate will be regenerated automatically when one of the following occurs:

- The `ixload_certificate.crt`, `ixload_privkey.key` or `san.log` files is deleted.
- The certificate has expired (it has a duration of 10 years).
- One of the entries required for SNI/SAN changes or disappears. For example, an IP address is changed, a hostname is changed, or a network interface disappears.

Script Changes Required for HTTPS

The IxLoad REST scripts samples have been updated to support HTTPS requests over IxLoad REST API.

The changes are:

- `kGatewayPort = 8443` – changed from 8080 to 8443
- `kResourcesUrl = 'https://%s:%s/api/v0/resources'` – changed from http to https

The utility files used by the IxLoad REST scripts samples have also been updated accordingly.

In `Utils\IxRestUtils.py`, the changes are:

- `connectionUrl = "https://%s:%s/" % (server, port)` – changed from http to https
- `result = self._getHttpSession().request(method, absUrl, data=str(data), params=params, headers=headers, verify=False)`

The `verify` parameter is a parameter provided by the `requests` library that is used in the REST scripts to generate HTTP/HTTPS requests. This parameter can take three values:

- `False`, as specified in the example above. If set to `False`, the HTTPS request will not perform any validation against a certificate.
- `True`, in this case, the HTTPS request will perform a validation against a set of predefined certificate bundles specific to the Python `requests` module.
- `'<certificate_path>'`. In this case, the HTTP request will perform a validation against the certificate specified at the path provided in the `verify` parameter.

To provide the certificate path, you must copy the certificate from the machine where the IxLoad Gateway is installed to the machine where the REST script is executed. The location where the certificate is copied will be provided as the certificate path.

If the certificate is regenerated and the verify parameter is set to a certificate path in a REST script on a remote machine, then that certificate will have to be downloaded again.

Errors from REST UI Clients

If you use a REST UI client such as Postman or Advanced REST Client, trying to access a URL from the IxLoad REST API might not work at first. This is because these two applications are tightly coupled to the Google Chrome browser. To be able to access any URL from the IxLoad REST API, you must first access one URL from the Google Chrome browser, accept the exception thrown by the browser (since the web server is using a self-signed certificate) and then proceed to using the REST client.

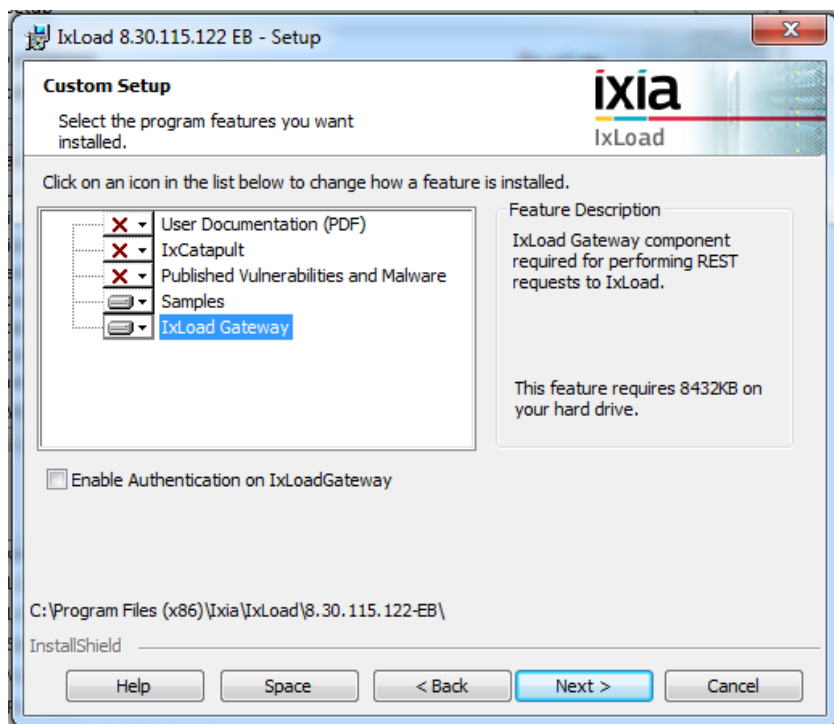
REST Authentication

Starting with the IxLoad 8.30 Update 2 release, you can use the IxLoad REST API with authentication. After enabling authentication, most REST requests must include a unique user api-key. For the current release this functionality is optional, integrated with Ixia User Management defined users only, and available for use with IxLoad versions running on Windows. REST Authentication is not available when using IxLoad on Linux.

Prerequisites

Enabling Authentication

To enable REST Authentication, you must select Custom Setup during the IxLoad install process and choose the IxLoad Gateway feature. After selecting the IxLoad Gateway, check the **Enable Authentication on IxLoadGateway** checkbox to enable authentication.



The authentication feature can be enabled or disabled every time IxLoad and IxLoadGateway are installed. This means that if you install one IxLoad/IxLoadGateway version and enable authentication, then you install a newer version and you do not check the 'Enable Authentication on IxLoadGateway' checkbox, after the install is completed, authentication will be disabled.

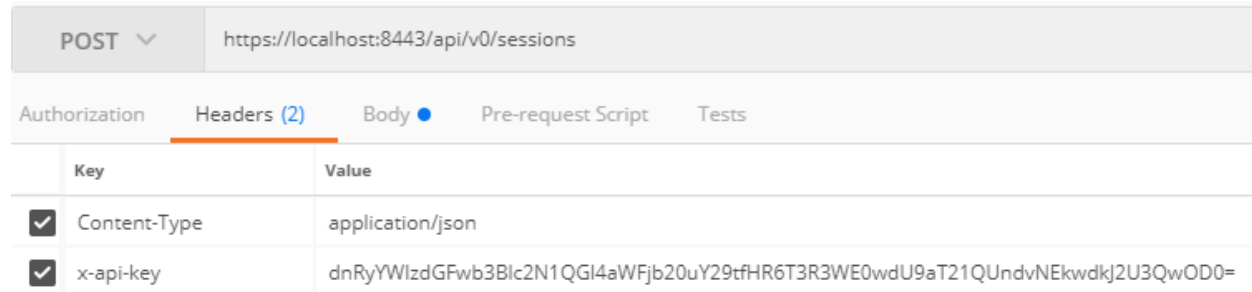
Ixia User Management

Another prerequisite is that an instance of an Ixia User Management server needs to be configured and present in the network, since for the current release REST Authentication is only supported in conjunction with Ixia User Management defined users.

User Management is a standalone application that you can download from the IxLoad section of Ixia's website (<https://support.ixiacom.com/support-overview/product-support/downloads-updates/versions/33>).

Authenticating REST Requests

With 8.30 Update 2, an 'api-key' has been added to the request headers. This will be generated by the UserManagement component based on a (username, password) pair. As a result, most requests will need to have an api-key present in their headers (see below for the exceptions). An example can be seen in the screenshot bellow taken from the REST UI Client Solution Postman.



Key	Value
<input checked="" type="checkbox"/> Content-Type	application/json
<input checked="" type="checkbox"/> x-api-key	dnRyYwIzdGFwb3Blc2N1QGI4aWFJb20uY29tfHR6T3R3WE0wdU9aT21QUndvNEkwdk2U3QwOD0=

The only requests allowed without including an api-key in the requests are:

- Getting the list of all created sessions: GET <https://localhost:8443/api/v0/sessions/>
- Getting the general status of a particular session: GET <https://localhost:8443/api/v0/sessions/1>

All other session-specific operations will require the presence of an api-key.

Once a session is created, the api-key provided is validated against the Ixia User Management database through the User Management server. If the key is not valid, an appropriate message will be returned.

As part of all the other requests that manipulate a session, the api-key provided is compared with the api-key used to create that particular session.

Possible results when executing a request are:

- If the operation was successful, a Status: 201 Created or 200 OK will be received;
- If the api-key was not specified in the headers, a Status: 403 Forbidden will be received, with the following message:

```
{
  "status": "POST operation failed",
  "error": "X-API-Key is not included in the header"
}
```

- If the api-key provided is not valid (does not exist in the UserManagement database), a Status: 403 Forbidden will be received, with the following message:

```
{
  "status": "POST operation failed",
  "error": "The provided X-API-Key is not valid"
} (this response is possible only for the CREATE session operation)
```

- If the api-key is not valid for a session (not the same as the one used to create the session), a Status: 403 Forbidden will be received, with the following message:

```
{
  "status": "POST operation failed",
  "error": "X-API-Key mismatch"
}
```

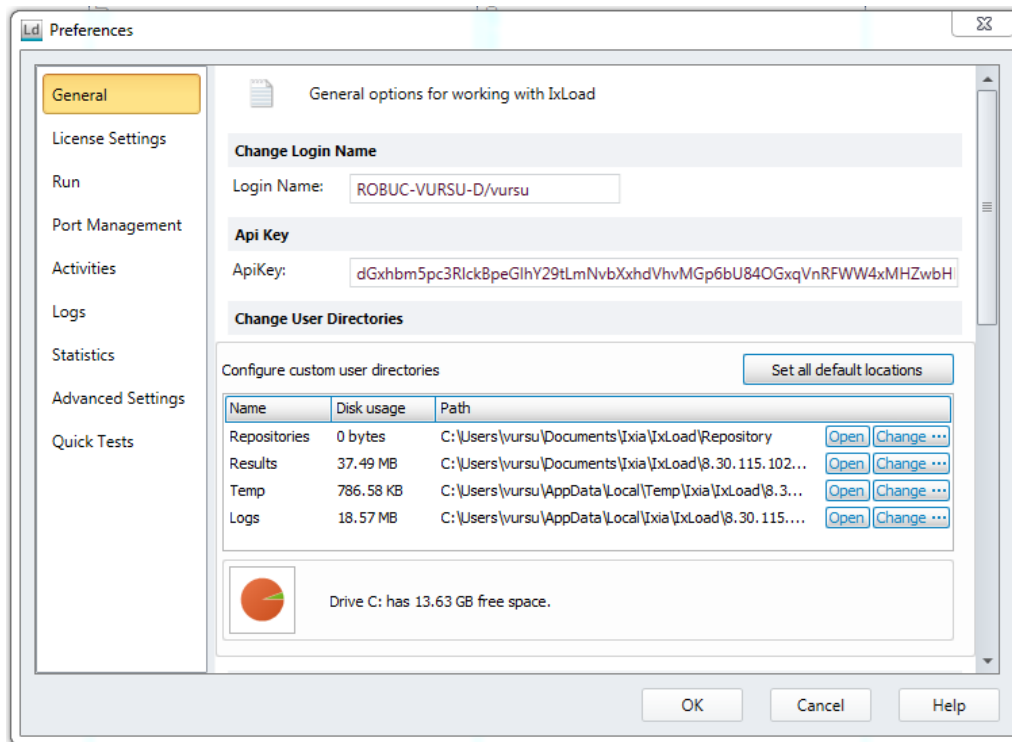
A user can delete only his/her own sessions (sessions that were created with the same api-key as the one provided during the DELETE request).

Retrieving the api-key

You can retrieve an api-key from the IxLoad UI.

In Ixload, when authentication is enabled and you log in with your Ixia User Management credentials, the api-key value can be retrieved from File->Preferences->General .

The value of the api-key will automatically update its value every time you change your password or another user logs in. The field is read-only, so you can only copy the value of the api-key, but not modify it.



Script Changes Required for Authentication

The changes that need to be made to IxLoad REST scripts for authentication are:

- `kApiKey = ""` – if this value remains the empty string, the api-key will not be included in the header of the requests; otherwise it will be part of it;
- `connection.setApiKey(kApiKey)` - setting the api-key for the connection

Supported methods and running operations

The IxLoad REST API is defined by how the resources are represented, by how they are accessed and changed and by the exposed data model.

REST representation

The Ixload REST api will handle a lot of different object types. Each object will have amongst its values:

- Primitive values, these are basic values;
- Complex values, these will be represented by lists or other REST resources.

Primitive values

Primitive values (numbers, string, bool) will be used as values for rest options in the requests payload. These should be represented as follows:

- strings will be enclosed in quotes : "custom string", ""
- numbers, integer or float will not be enclosed in quotes : 1 , 1.1
- Booleans will not be enclosed in quotes, and will start with lowercase : true, false

List Objects

The IxLoad data model contains numerous lists. In order to be able to identify a resource that is part of a list (it must have a unique URL), the resource must have an ID associated, that is unique in the containing list. For this reason, each resource that is contained in a list has a field that contains its ID. This field is called 'objectID' in IxLoad. However, this name can be retrieved programatically by performing an 'OPTIONS' request on the resource, and retrieving the value for the 'resourceIdName' field - right now this will return 'objectID'.

A resource's objectID can be retrieved by performing a GET request on the list, and iterating through the results, each element of the list (each resource) will have this field set.

For a list that has the following URL:

- <http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList>

An element with objectID = 10 will be retrieved the following URL:

- <http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList/10>

The requests listed in the below screenshots were made from the Google Chrome 'Advanced REST Client' add-on.

IxLoad REST API supports the following HTTP methods:

REST resources

Other REST resources will be shown as links to another object. So each time an object will be retrieved through the REST API, it may have primitive values, lists and other REST objects. The other REST objects will be shown as links that will point to the data model location of the referenced REST object.

Case conventions

In IxLoad REST API URLs are case insensitive. This applies to all parts of any URL, with the exception of the 'api' string at the beginning of the URL. This is not the case, however, for fields and values entered in request payloads. The field names entered in the payload are actually option names in IxLoad middleware, so the case defined must be followed.

Preferences

You can change several Global Options directly from the REST API using the following URL:

- <http://127.0.0.1:8080/api/v0/sessions/0/ixload/preferences>

The options that can be changed are shown in the following image:

```

1 {
2   "continueTestOnLoadModuleFail": true,
3   "logCollectorSize": 100,
4   "links": [{"name": "links"}],
10  "maximumInstances": 3,
11  "enableDebugLogs": false,
12  "overloadProtection": true,
13  "autoRebootCrashedPorts": false,
14  "detailedChassisMonitoring": false,
15  "licenseModel": "Subscription Mode",
16  "checkLinkStateAtApplyConfig": true,
17  "ntpServer2": "10.215.170.83",
18  "ntpServer1": "0",
19  "csvThroughputUnits": "Bps",
20  "allowIPOverlapping": false,
21  "allowRouteConflicts": true,
22  "restObjectType": "ixRestPreferences",
23  "enableAnonymousUsageStatistics": false,
24  "licenseServer": ""
25 }
```

Note: IxLoad REST API sessions are started under the System user, not the user you are logged in as. As all the Global Options except Maximum Instances, License Model, and License Server are saved per-user, this means that settings made in the IxLoad UI will have no effect on REST API runs, since the REST API will be registered under the 'System' user. Therefore, for the Maximum Instances, License Model, and License Server options to have an effect on REST API tests, you must set them from the REST API.

These options can be changed by performing PATCH requests on the 'preferences' URL, with a payload as below:

```
{"licenseServer": "ipOrHostname"}
```

IxLoad REST Methods

The IxLoad REST API supports the following HTTP methods - GET, PATCH, POST, DELETE, OPTIONS. The only content type we will support for payloads will be JSON. The Payload applies to PATCH, POST and DELETE methods.

GET

Users can make GET requests in order to receive the list of REST options for the requested resource. The GET request will not contain any payload. If the request is successful, a '200 OK' status will be returned.

The result will be a JSON dictionary containing the option names and values exposed by the resource. All the primitive options (bool, string, int) will be in the root dictionary, while complex options (other objects) will be placed together, as a list, under the 'links' option. Each element of the 'links' list will be a dictionary that contains the following:

- 'rel' : the child resource name
- 'href' : a URL at which the child resource can be accessed

Figure 1 shows the output of a GET method applied to the activeTest REST resource in IxLoad.

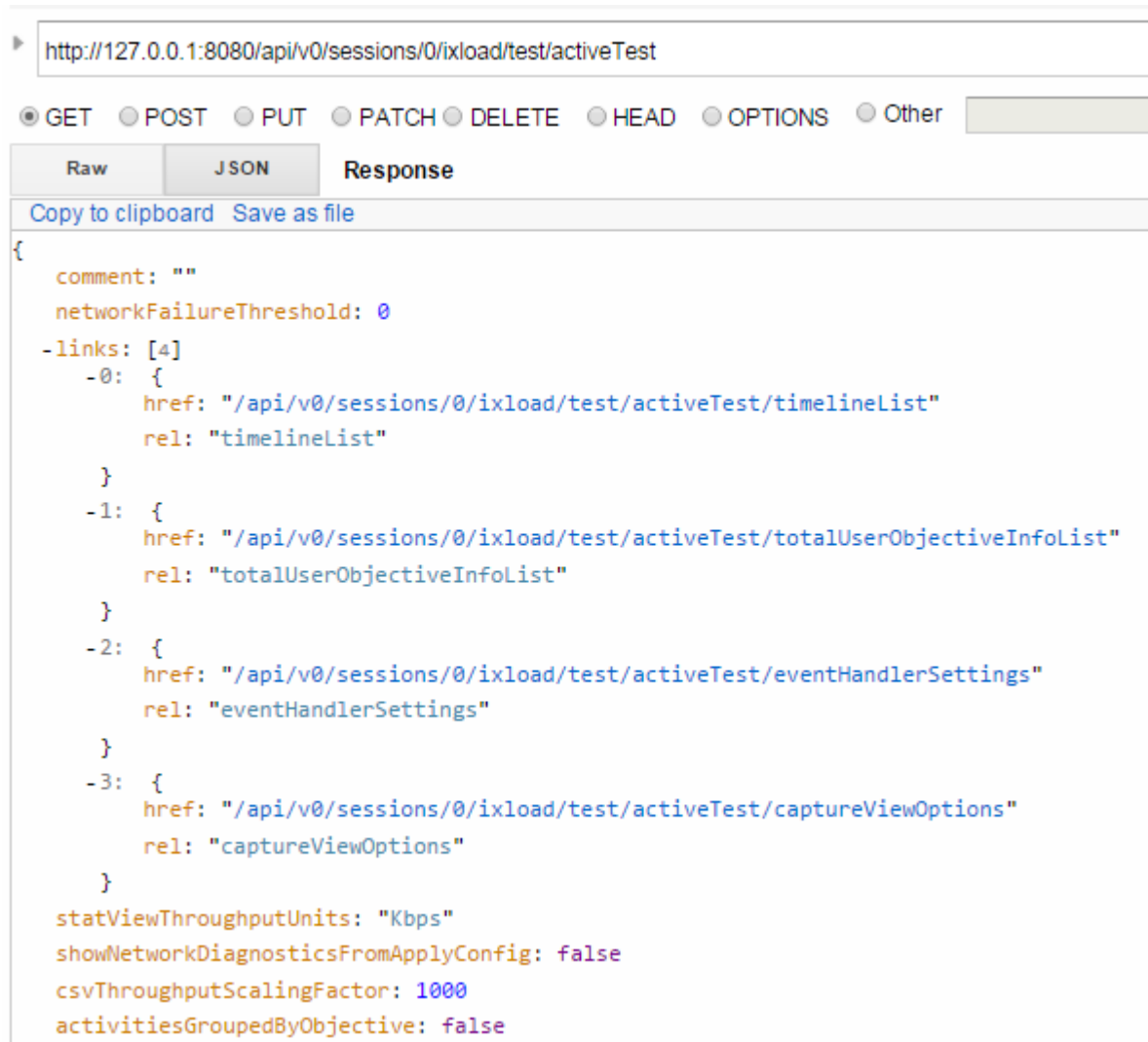


FIGURE 1 GET METHOD

The above representation is only a demonstration of how the output of GET is provided in IxLoad REST API through the use of the Advanced REST Client in Google Chrome. The actual representation will be different according to the programming language that will be used to access the IxLoad REST API.

PATCH

Users can perform PATCH requests in order to change field values on resources exposed by the IxLoad session. The PATCH request will receive as payload a list of options that the user wishes to modify. Each pair in the dictionary will contain a field name and the new option for it. If the request is successful, a '204 No Content' status will be returned.

The payload for a PATCH request must contain at least one field that will be changed. This means one "field name": "new value" pair. Figure 2 shows the representation of the PATCH method in Advanced REST Client.

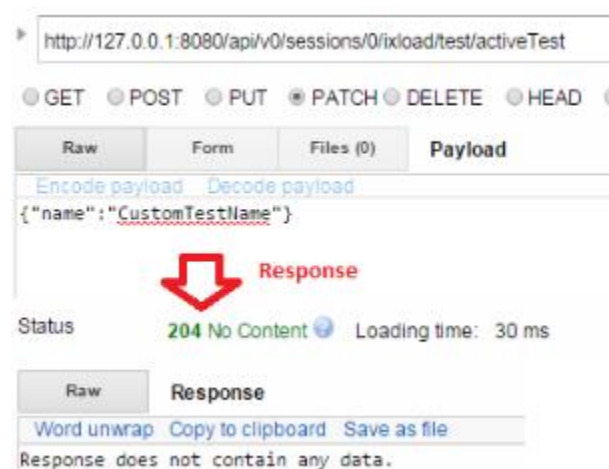


FIGURE 2 PATCH METHOD

Most resources cannot be modified using PATCH requests while a test is running. If a PATCH request is made while a test is configuring / running, a '400 Bad Request' status will be returned.

```
{
  status: "PATCH operation failed"
  error: "Cannot change HTTPClient1 at this moment. Please try again later"
}
```

POST

Users can make POST requests in order to add elements to a list. The request is made on the list url , and the actions that take place behind the scenes are to instantiate a new object of the type given by the list and then add the newly created object to the list. If the request is successful, a '201 Created' status will be returned.

The payload for a POST request will represent the parameters used when creating the resource that will be added to the list. Since not all resources (objects) require parameters in the constructor, the payload for a POST request can be empty ({}).

Figure 3 Shows the output of the POST method in the Advanced REST Client.

The screenshot shows a REST client interface with the following details:

- URL:** `http://127.0.0.1:8080/api/v0/sessions`
- Method:** POST (selected)
- Headers:** (Empty)
- Payload:** `{"ixLoadVersion": "8.00.0.195"}`
- Content-Type:** application/json
- Status:** 201 Created (circled in green), loading time: 9 ms
- Request headers:**
 - User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36
 - Origin: chrome-extension://hgmlfoofddfdnphfgcellkdfbfjeloo
 - Content-Type: application/json
 - Accept: */*
 - Accept-Encoding: gzip, deflate
 - Accept-Language: en-US,en;q=0.8
 - Cookie: JSESSIONID=6F4A7F28464E06921C8784F490B8A464
- Response headers:**
 - Date: Fri, 13 Nov 2015 15:19:57 GMT
 - Content-Length: 2
 - Content-Type: application/json
 - Location: /api/v0/sessions/0
 - Server: CherryPy/3.6.0

FIGURE 3 POST METHOD

In the response headers there will be a field called 'Location', which will contain the URL address of the newly created object

Elements cannot be added to a list while a test is running. If a POST request is made while a test is configuring / running, a '400 Bad Request' status will be returned.

```
{
  status: "POST operation failed"
  error: "Cannot perform the 'POST' operation at this moment. "
}
```

DELETE

Users can make DELETE requests in order to delete one (or all) of the elements of the list. If the request is successful, a '204 No Content' status will be returned.

DELETE requests don't require any payload.

If the DELETE request is made on a list URL, the list will be cleared - all the elements will be removed.

If the DELETE request is made on a URL that consists of the list URL and an object's unique ID appended at the end, only the object with that objectID will be removed.

Example 1: DELETE on <http://127.0.0.1:8080/api/v0/sessions> will delete all sessions

Example 2: DELETE on <http://127.0.0.1:8080/api/v0/sessions/2> will only delete the session with objectID = 2

Elements cannot be removed from a list while a test is running. If a DELETE request is made while a test is configuring / running, a '400 Bad Request' status will be returned.

```
{  
  status: "DELETE operation failed"  
  error: "Cannot perform the 'DELETE' operation at this moment."  
}
```

OPTIONS

Users can make OPTIONS requests on any resource. The output of the request will be a set of information about the product and resource properties. If the result is successful, a '200 OK' status will be returned.

OPTIONS requests don't require any payload.

As previously stated, in the OPTIONS response there will be two fields that specify the names of the unique object id field and the name under which all complex resources are kept on GET requests (the 'links' option name).

Figure 4 shows how the OPTIONS method output looks in the Advanced REST Client.



The screenshot displays a REST client interface. At the top, the URL `http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/eventHandlerSettings` is entered. Below the URL, several HTTP methods are listed with radio buttons: GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS (which is selected), and Other. A red arrow points from the OPTIONS method to the status bar. The status bar shows `200 OK` and a loading time of `23 ms`. Below the status bar, there are tabs for `Raw`, `JSON`, and `Response`. The `Response` tab is active, showing a JSON object with the following structure:

```
{
  -product: {
    version: "1.0.0.0"
    name: "eventhandlersettings"
    custom: null
  }
  -properties: [4]
    0: "disabledEventClasses"
    1: "disabledPorts"
    2: "objectID"
    3: "objectType"
  -features: {
    -rest: {
      multipost: false
      multidelete: true
      put: false
      patch: true
      typeName: "objectType"
      resourceIdName: "objectID"
      maxlist: null
      linksName: "links"
    }
    -session: {
      supported: true
      multiApp: true
    }
    -queryParam: {
      defaultEmbeddedValue: false
      embedded: false
      deepchild: false
      links: false
      includes: true
    }
    -auth: {
      authType: null
    }
  }
}
```

FIGURE 4 OPTIONS METHOD

Operations

Besides the HTTP requests listed above, executed on basic resources (objects or lists of the IxLoad data model), IxLoad REST API also offers support for operations. These are asynchronous actions performed on a certain resource (URL), that do not add, remove or change field values for the resource they are applied to, but rather change its state. Examples of operations are starting an inactive IxLoad session, connecting to an already existing chassis or running a test.

In order to check when operations are available for a certain resource, a GET request can be performed on the resource URL, adding '/operations' at the end of the URL.

Figure 5 shows how the operations available for the test REST resource are represented.



FIGURE 5 OPERATIONS GET

The available operations will be listed in the response, containing both the operation names and the parameters they require. Default values for parameters will also be shown.

Starting an operation

Starting an operation is done by performing a POST request on the following URL:

`$resourceUrl/operations/$operationName`. The request payload represents the parameters required by the operation, as shown in the screenshot above. Some operations (such as `runTest`) may not require any parameters, so for them an empty payload must be sent: `{}`.

Figure 6 Shows the output of the post command for the `loadTest` operation.

http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/operations/loadTest

GET POST PUT PATCH DELETE HEAD OPTIONS

Raw Form Files (0) Payload

Encode payload Decode payload

`{"fullPath": "stats.rxf"}`

Status: 202 Accepted Loading time: 18 ms

Request headers: User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/42.0.2866.90 Safari/537.36; Origin: chrome-extension://hgmlfoofddfdnphfgcellkdfbfjeloo; Content-Type: application/json; Accept: */*; Accept-Encoding: gzip, deflate; Accept-Language: ro-RO,ro;q=0.8,en-US;q=0.6,en;q=0.4

Response headers: Date: Tue, 20 Oct 2015 21:00:12 GMT; Content-Length: 2; Content-Type: application/json; Location: api/v0/sessions/0/ixload/test/operations/loadTest/1; Server: Caddy/0.9.0

FIGURE 6 OPERATIONS POST

uploadFile

```
uploadFile(connection, url, fileName, uploadPath, overWrite)
```

This operation uploads a file from the machine where the script runs on the machine where the IxLoad client is running.

`connection` is the connection object that manages the HTTP data transfers between the client and the REST API.

`url` is the address of the resource that uploads the file. This url should be in form of:

```
http://ip:port/api/v0/resources.
```

`filename` contains the name (or absolute path to the file, if the file is not in the same location as the executing script) of the file to be uploaded. This is the location on the machine where the script is running. Example: `"file.txt"`, `r"D:\\examples\\file.txt"`.

`uploadPath` is the path where the file should be copied to on the machine on which the IxLoad client runs.

`overWrite` specifies the desired behavior if the file to be uploaded already exists on the remote machine. The default value is `'True'`.

Getting an Operation's Status

Since these operations are asynchronous methods, users must be able to check an operation's status after they have started it. For this, when starting an operation (executing the POST request), in the response header there will be a field called 'Location', that will contain a URL. While performing a GET request on that URL, the operation's status will be returned. Figure 7 shows the output for getting the operation status belonging to the loadTest operation.

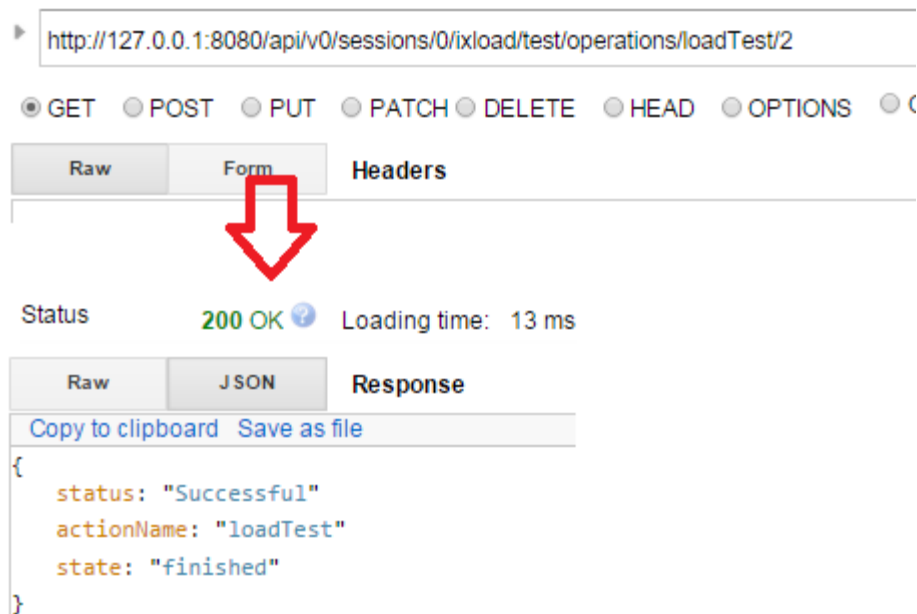


FIGURE 7 GET OPERATION STATUS

Possible values for the 'state' field are:

- Created – implies the operation was created;
- Executing – implies the operation is in progress;
- Finished – implies operation is done;

Possible values for the 'status' field are:

- Not started – implies the operation has not started yet, as operations are synchronous probably it is waiting for other operations to finish processing;
- In Progress – implies the operation is executed;
- Successful – implies the operations has finished and was successful;
- Error – implies the operation has finished but was not successful because an error has occurred.

In case the operation will fail (exit with an error), in the response above a new field will be introduced, that will contain the error message returned by the operation:

```
{
  status: "Error"
  actionName: "loadTest"
  state: "finished"
  error: "File doesn't exist - F:\statsdfs.rxf"
}
```

Important: the URL where an operation's status can be retrieved is only active for a certain amount of time. At the moment, this lifetime is of 10 minutes. If a GET request on the `operationUrl/operationID` URL is performed after this period, the status will not be retrieved, but a '400 Bad Request' error will be returned.

Examples of Common Operations in the IxLoad REST API

A list of the most commonly used operations for an IxLoad test in the REST API can be obtained by doing a GET on <http://localhost:8080/api/v0/sessions/0/ixload/test/operations> . The result will list the following operations (also shown in the image below):

- **loadTest** – Load an IxLoad configuration file. The fullPath of the rxf to be loaded will need to be passed on as a parameter.
- **importConfig** - imports a .crf file as the current test configuration. The location of the .crf file and the location where the .rxf file will be saved after the import need to be passed as parameters.
- **saveAs** – Save the currently-loaded configuration file as a new file. The new file path for the rxf will need to be passed as a parameter, and the overwrite option in case the file path already exists.
- **save** – Save the currently loaded configuration file.
- **applyConfiguration** – Apply configuration on the current IxLoad test. The test will go to Configured state. This is equivalent to clicking the Apply Config button in the IxLoad UI.
- **runTest** – Run the current IxLoad test. The test will go to running state directly. This action is equivalent clicking the Run test button in the IxLoad UI.
- **waitForAllCaptureData** - waits for the test to capture all the port data that was received after the test has finished running.
- **abortAndReleaseConfigWaitFinish** – Stop the currently running IxLoad test.
- **exportConfig** – Exports the currently loaded configuration file as a .crf file. The location of the archive needs to be passed as a parameter.

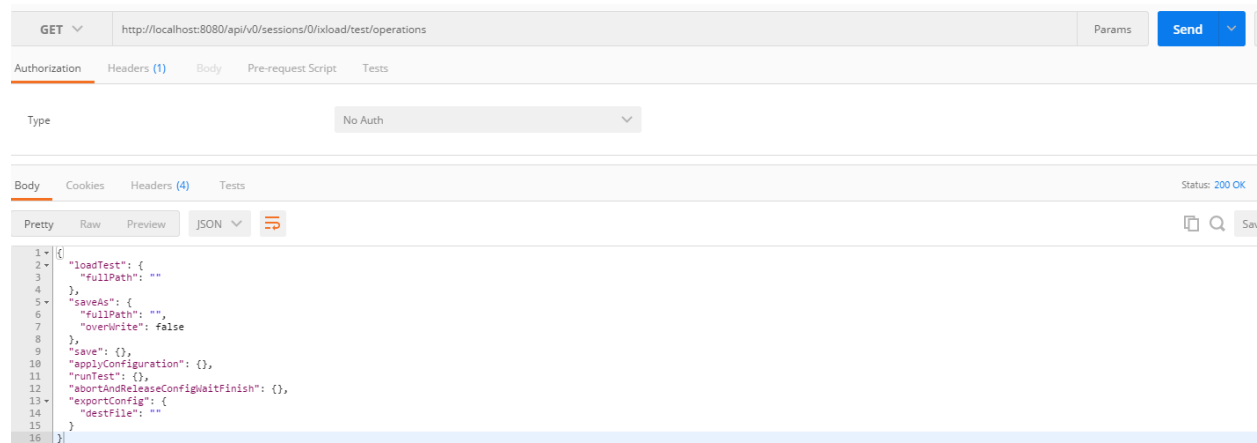


FIGURE 8 LIST OF OPERATIONS

Example for Loading a Repository (.rxf) File

On an already created and activated session do a POST on an URL similar to:

[http://localhost:8080/api/v0/sessions/\[SESSIONID\]/ixload/test/operations/loadTest/](http://localhost:8080/api/v0/sessions/[SESSIONID]/ixload/test/operations/loadTest/)

In the payload/body of the request, add the path to the .rxf file.

```
{"fullPath": "C:\\http_test.rxf"}
```

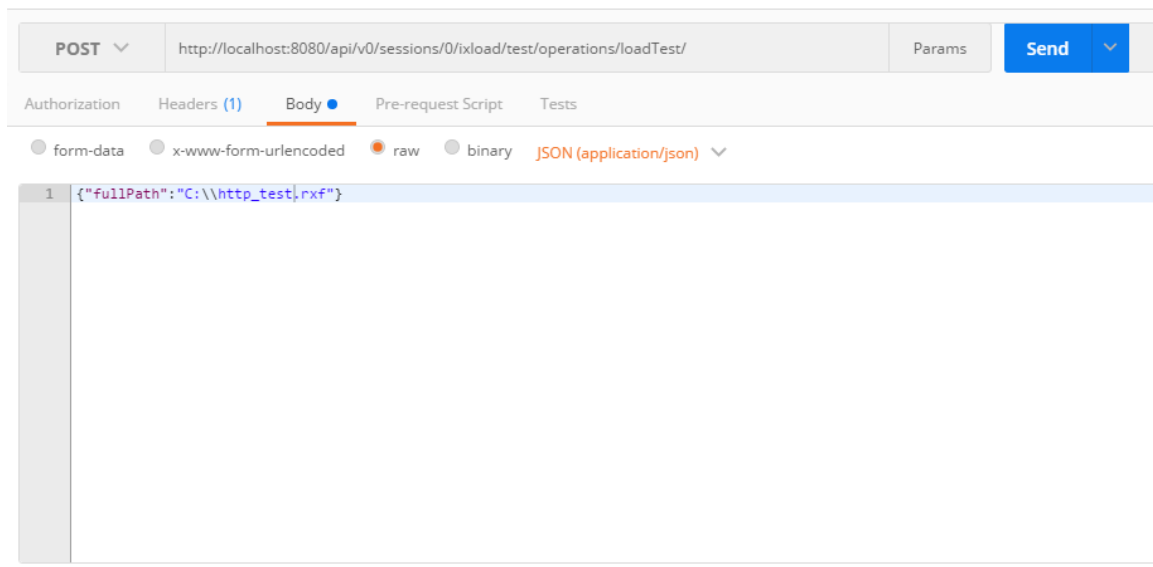



FIGURE 9 PATH TO RXF FILE

As described in Getting an Operation's Status, query the status of the operation until the state is Finished.

Example for Importing a .crf File

On an already created and activated session do a POST on an URL similar to:

[http://localhost:8080/api/v0/sessions/\[SESSIONID\]/ixload/test/operations/importConfig](http://localhost:8080/api/v0/sessions/[SESSIONID]/ixload/test/operations/importConfig)

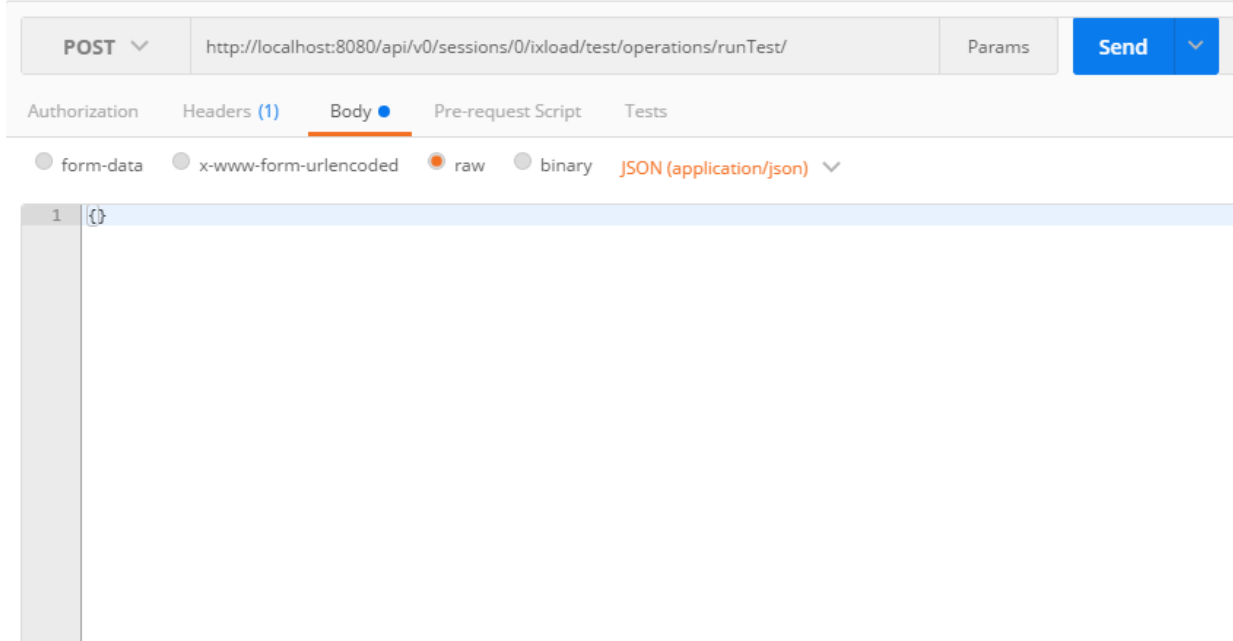
In the payload/body of the request, add the path to the .rxf file:

```
{"srcFile": "C:\\mycrf.crf", "destRxf": "C:\\rxf_from_crf.rxf"}
```

Example of Running a Test

On an already created and activated session in which there is either a loaded configuration file or a new test has been created, do a POST on a URL similar to:

[http://localhost:8080/api/v0/sessions/\[SESSIONID\]/ixload/test/operations/runTest/](http://localhost:8080/api/v0/sessions/[SESSIONID]/ixload/test/operations/runTest/)

**FIGURE 10** RUNNING A TEST

As described in Getting an Operation's Status, query the status of the operation until the state is Finished.

Example for Waiting to Capture the Port Data

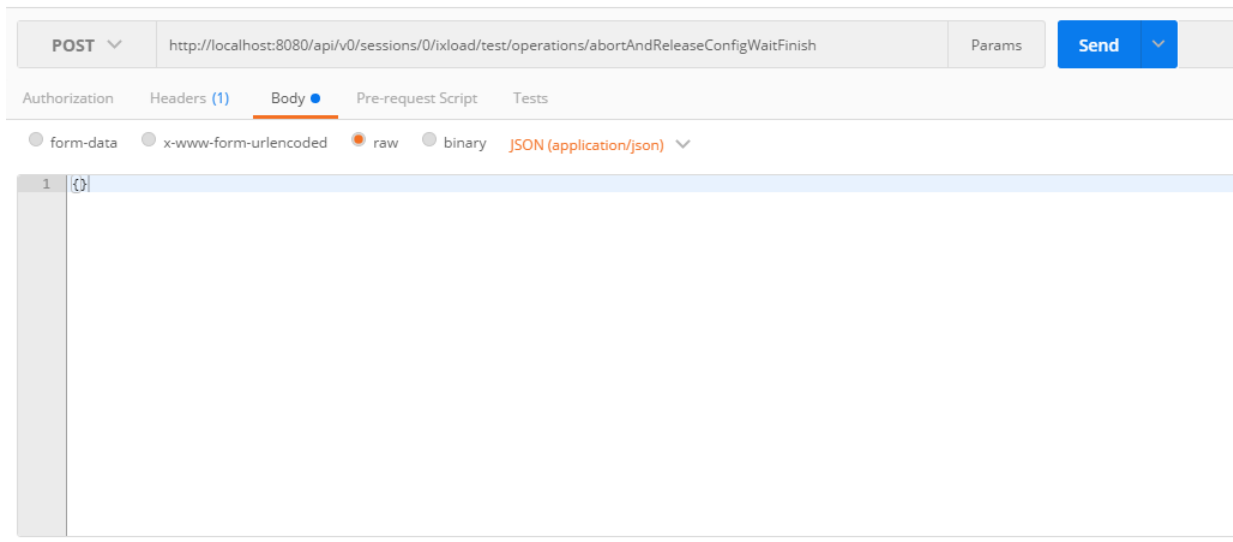
On an already created and activated session do a POST on an URL similar to:

[http://localhost:8080/api/v0/sessions/\[SESSIONID\]/ixload/test/operations/waitForAllCaptureData](http://localhost:8080/api/v0/sessions/[SESSIONID]/ixload/test/operations/waitForAllCaptureData)

Example of Stopping a Test

On an already created and activated session in which there is either a loaded configuration file or a new test has been created, do a POST on a URL similar to:

[http://localhost:8080/api/v0/sessions/\[SESSIONID\]/ixload/test/operations/abortAndReleaseConfigWaitFinish](http://localhost:8080/api/v0/sessions/[SESSIONID]/ixload/test/operations/abortAndReleaseConfigWaitFinish)

**FIGURE 11** STOPPING A TEST

As described in Getting an Operation's Status, query the status of the operation until the state is Finished.

Query Strings

You can search using a filter with one or more parameters separated by commas.

Note: The format for Query Strings changed between release 8.00 and 8.10:

For 8.00, the format was: <http://resourceUrl?fieldName=value>.

Beginning with 8.10, the format is [http://resourceUrl?filter=""fieldName <operator> value](http://resourceUrl?filter=)

The query strings are inserted under the 'filter' param, at the end of the URL. The supported query string operators are:

- eq - equals
- ne - not equal to
- lt - lower than
- gt - greater than
- le - lower or equal to
- ge - greater or equal to

When the 'eq' operator is used for string fields (like names of statistics, for example), it automatically has a 'contains' effect. This means that a GET request on `/configuredStats?filter=""caption eq HTTP"` will return all statistics whose caption contains "HTTP". If instead a 'matches' operation is desired, you can still use 'eq' but the value must be enclosed in quote marks (""). This will cause a GET on `/configuredStats?filter=""caption eq "HTTP""` to return only statistics whose caption is exactly "HTTP".

Multiple query string conditions can be introduced in the same URL, separated by commas.

For example, the following URL will return all enabled statistics whose objectID is less than or equal to 14:

- [http://localhost:8080/api/v0/sessions/0/ixload/stats/HTTPClient/configuredStats?filter=""enabled eq True,objectID le 14"](http://localhost:8080/api/v0/sessions/0/ixload/stats/HTTPClient/configuredStats?filter=)

Query Strings are only supported on list resources, with the following methods:

- GET: Returns all the elements of the list which satisfy the query string conditions.
- PATCH: Modifies the list with the parameters sent in the request payload all the elements of the list which satisfy the query string conditions.
- DELETE: Deletes from the list all the elements of the list which satisfy the query string conditions.

Collecting Diagnostics

IxLoad includes a diagnostics collection utility that collects log files and packages them into a ZIP file, so that they can be stored or emailed conveniently. In the GUI, the utility can be accessed from the File > Tools > Diagnostics menu. You can collect those same log files using the REST API.

To collect diagnostics:

- At least one session must be active.
- The test must be in either the Configured or Unconfigured state.

To collect diagnostics, use the following command:

[POST @ api/v0/sessions/72/ixload/test/activeTest/operations/collectDiagnostics](http://localhost:8080/api/v0/sessions/72/ixload/test/activeTest/operations/collectDiagnostics)

Specify the ZIP file location as the POST payload:

```
{"zipFileLocation": "<path to save ZIP file>"}
```

For example:

```
{"zipFileLocation": "C:\\Users\\ixia\\Desktop\\diags.zip"}
```

Figure 12 shows an example of a POST operation to collect diagnostics from a REST client.

The screenshot displays a REST client interface for a POST request. The URL is `http://127.0.0.1:8080/api/v0/sessions/72/ixload/test/activeTest/operations/collectDiagnostics`. The request method is POST, and the content type is `application/json`. The raw payload is `{"zipFileLocation": "C:\\Users\\ixia\\Desktop\\diags.zip"}`. The response status is `202: Accepted` with a loading time of 27 ms. The response headers include `Date: Mon, 01 Aug 2016 08:50:50 GMT`, `Content-Length: 2`, `Content-Type: application/json`, `Location: api/v0/sessions/72/ixload/test/activeTest/operations/collectDiagnostics/1`, and `Server: CherryPy/3.6.0`. The response body is empty JSON `{}`.

FIGURE 12 COLLECTING DIAGNOSTICS

The status of the POST operation to collect diagnostics should be `202: Accepted`. The response to the operation should include a location.

To query the status of the POST operation, use a GET operation and specify the location received in the response to the POST.

For example:

[GET @ http://127.0.0.1:8080/api/v0/sessions/72/ixload/test/activeTest/operations/collectDiagnostics/1](http://127.0.0.1:8080/api/v0/sessions/72/ixload/test/activeTest/operations/collectDiagnostics/1)

Figure 13 shows an example of a query to get the status of a diagnostics collection operation.

<http://127.0.0.1:8080/api/v0/sessions/72/ixload/test/activeTest/operations/collectDiagnostics/3>

GET
 POST
 PUT
 DELETE
 Other methods

Raw headers Headers form Headers sets

Content-Type: application/json

Status: 200: OK ? Loading time: 25 ms

Response headers (4) Request headers (1) Redirects (0) Timings

Date: Mon, 01 Aug 2016 09:03:59 GMT
 Content-Length: 118
 Content-Type: application/json
 Server: CherryPy/3.6.0

Raw JSON

```

{
  "status": "In Progress"
  "actionName": "collectDiagnostics"
  "state": "executing"
  "result": ""
}
  
```

FIGURE 13 GETTING THE STATUS OF THE DIAGNOSTICS COLLECTION OPERATION

IxLoadGateway - IxLoad session handling

Creating and handling IxLoad sessions is done through an IxLoad service, named IxLoadGateway.

IxLoadGateway is installed with IxLoad as part of the custom install options.

Creating a new session

In order to create a new session object a POST on `api/v0/sessions` with payload: `{"ixLoadVersion": "version no."}` Needs to be performed.

Please note that this means the session is just created not started and not active.

This will not take into consideration the instance count limit on the client side. This will only work for sessions that are started. Figure 14 shows how this looks from rest client.

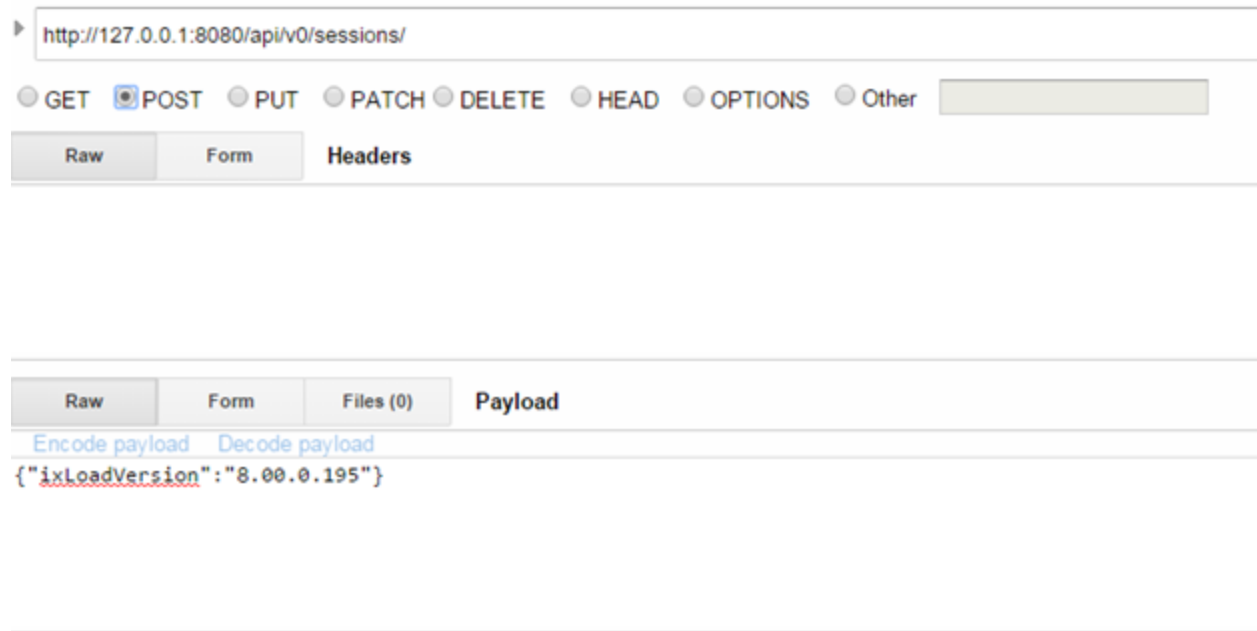


FIGURE 14 CREATE A NEW SESSION OBJECT

Figure 15 shows the response for the post above. Status is 201 created & location points to the session.

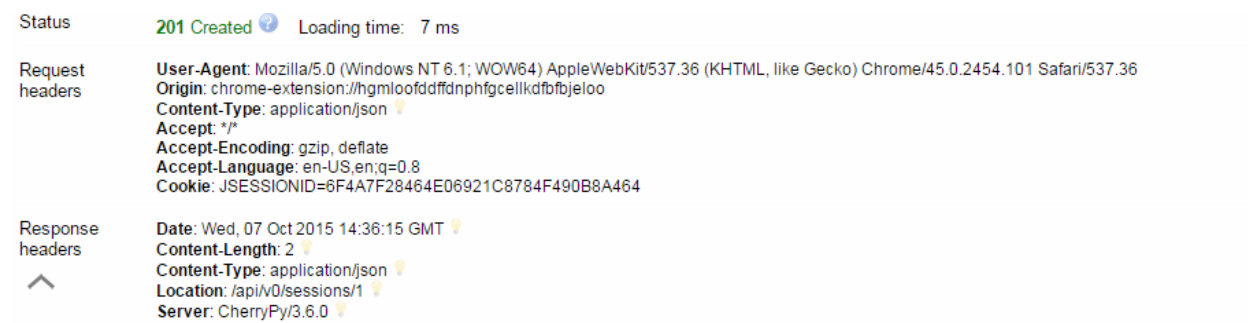


FIGURE 15 RESPONSE FOR CREATING A SESSION OBJECT

Starting a session

To start a session an operation is provided, named *start*. This operation is available on each individual session and requires no payload.

Figure 16 shows how this looks in rest client. This operation will start a new IxLoad session based on the IxLoad version for which the session was created.

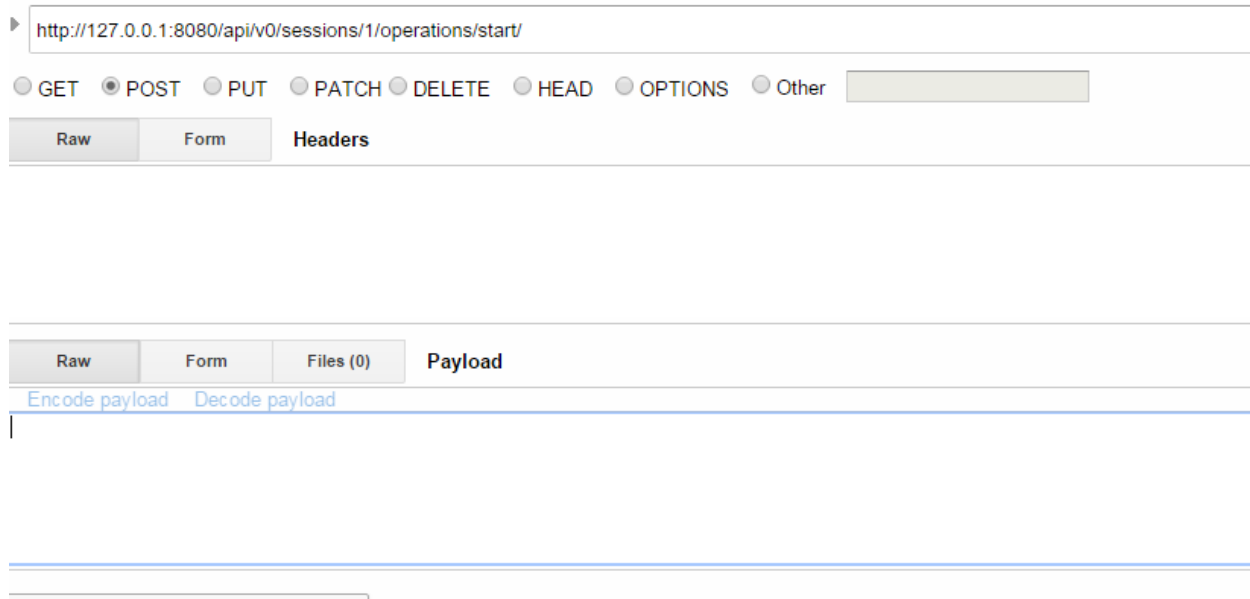


FIGURE 16 START AN EXISTING SESSION

Figure 17 shows the response for the start operation. Response is 202 Accepted. And the location shows the result for the operation.



FIGURE 17 RESPONSE FOR START OPERATION

Figure 18 shows how the operation result for start looks like when the session started successfully

It contains the same information as the now deprecated create operation.

http://127.0.0.1:8080/api/v0/sessions/1/operations/start/1

GET POST PUT PATCH DELETE HEAD OPTIONS Other

Raw Form Headers

Clear Send

Status **200 OK** Loading time: 8 ms

Request headers
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36
Content-Type: text/plain; charset=utf-8
Accept: */*
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Cookie: JSESSIONID=6F4A7F28464E06921C8784F490B8A464

Response headers
Date: Wed, 07 Oct 2015 14:45:20 GMT
Content-Length: 105
Content-Type: application/json
Server: CherryPy/3.6.0

Raw JSON Response

Copy to clipboard Save as file

```
{
  status: "Successful"
  actionName: "start"
  state: "finished"
  sessionId: 1
}
```

FIGURE 18 SUCCESSFUL START

Figure 19 shows the unsuccessful start operation due to maximum number of instances.

http://127.0.0.1:8080/api/v0/sessions/2/operations/start/2

GET POST PUT PATCH DELETE HEAD OPTIONS Other

Raw Form Headers

Status **200 OK** Loading time: 6 ms

Request headers
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36
Content-Type: text/plain; charset=utf-8
Accept: */*
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Cookie: JSESSIONID=6F4A7F28464E06921C8784F490B8A464

Response headers
Date: Wed, 07 Oct 2015 14:48:43 GMT
Content-Length: 179
Content-Type: application/json
Server: CherryPy/3.6.0

Raw JSON Response

Copy to clipboard Save as file

```
{
  status: "Successful"
  actionName: "start"
  state: "finished"
  errorMessage: "Already running maximum allowed copies of IxLoad."
  sessionId: 2
}
```

FIGURE 19 FAILED SESSION START

Deleting a session

Deleting an IxLoad session is done in the same way that was described for generic lists. A DELETE request can be sent either to the sessions list URL, or to an actual session URL. If the request is sent to the sessions URL, all sessions will be closed. If the request is sent to a specific session's object ID, only that session will be closed.

When deleting a session, the IxLoad process underneath it will be closed.

IxLoad Data Model

Using REST API users can browse the IxLoad data model in order to retrieve or modify the current configuration. The link below contains information on where to find in the data model the following resources: L47 plugins, L23 ranges, timelines. Also, the link contains supported operations, such as loading and saving configurations and running a test.

Communities

Users can find all the communities in the following path:

- <http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList/>

All the communities in the test will be shown in this list, regardless of their role - client / server / peer. Also, this will contain both enabled and disabled communities.

Users can choose to only view client communities by performing a GET operation on the same list, but using query strings:

- [http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList?filter="role eq client"](http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList?filter=\)

Community resources:

- **activities**: all the activities under a community can be found in the following list:
 - [http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList/\\$communityObjectID/activityList](http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList/$communityObjectID/activityList)
 - An activity's command list will be found under the 'agent' resource
- **port list**: the ports assigned to a community can be found on the 'network' resource under the community resource:
 - <http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList/0/network/portList>
- **IP ranges**: the IP ranges used by the community will be found under the 'network' resource:
 - <http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList/0/network/stack/childrenList/1/childrenList/1/rangeList>
 - 'stack' is the entry point in the L23 data model.

Timelines

All the timelines used in the test will show under a single list, located on the 'activeTest' resource:

- <http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/timelineList>

Plugins will not expose in REST a direct reference to their timeline (the activities don't have a 'timeline' option exposed), they will have a 'timelined' option. This option will contain the 'objectID' of the desired timeline in the test timeline list. If the user wants to change the timeline used by a certain plugin, he needs to perform a PATCH request on the activity with the following payload: {"timelined": "object ID of the desired timeline in the test timeline list"}

Login name

Users can change the login name that will be used by their session when running, by changing the 'loginName' field on the chassis chain resource:

- PATCH on <http://127.0.0.1:8080/api/v0/sessions/0/ixload/chassischain/>
- payload : { "loginName" : "NewLoginName" }

Modifying the Activity user Objective Value on the fly

While the test is running, the user can change the user objective value for an activity by performing a PATCH request on a URL similar to:

- <http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList/0/activityList/0>

with the following payload: {"userObjectiveValue": 100}

Chassis Chain / Port Assignment Operations

Through IxLoad REST API users can perform the following chassis / port operations:

- Add or remove a chassis
- Connect to a chassis
- Assign or un-assign ports

The chassis list can be found on the 'chassisChain' root object, at the following URL:

- <http://127.0.0.1:8080/api/v0/sessions/0/ixload/chassischain/chassisList>

Adding a chassis

Adding a new chassis will be done like below:

POST@ `api/v0/sessions/0/ixload/chassischain/chassisList` with `{"name":"chassis ip or name"}`

Figure 20 shows the input for rest client.

The newly added chassis is not connected and it has no cards or ports.

The screenshot shows a REST client interface with the following configuration:

- URL: `http://127.0.0.1:8080/api/v0/sessions/0/ixload/chassischain/chassisList`
- Method: **POST** (selected)
- Headers: **Raw** (selected), **Form**, **Headers**
- Payload: **Raw** (selected), **Form**, **Files (0)**, **Payload**
- Payload Content: `{"name": "10.215.170.77"}`
- Content-Type: `application/json` (selected), with a note: *Set "Content-Type" header to overwrite this value.*
- Buttons: **Clear**, **Send**

FIGURE 20 POST ON CHASSIS LIST

The response for this is shown in Figure 21.

The result is 201 Created.

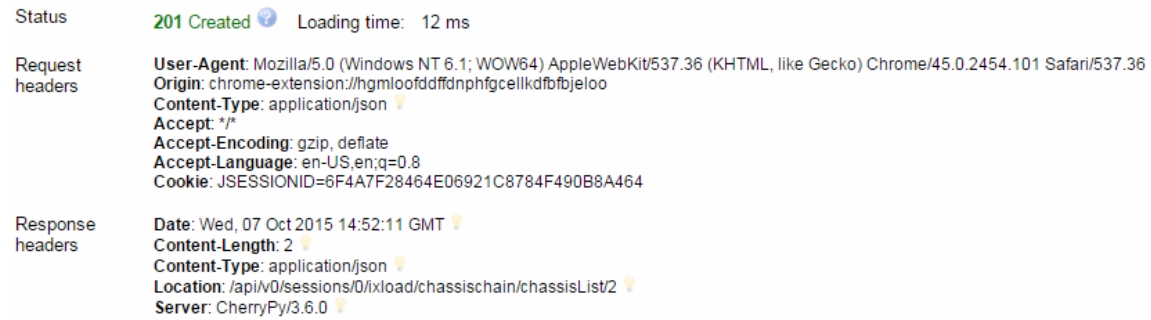


FIGURE 21 RESULT FOR POST ON CHASSISLIST

Connecting to a chassis

To connect to a chassis:

POST @ api/v0/sessions/0/ixload/chassischain/chassisList/2/operations/refreshConnection

No payload is required.

Figure 22 shows how this looks in rest client.

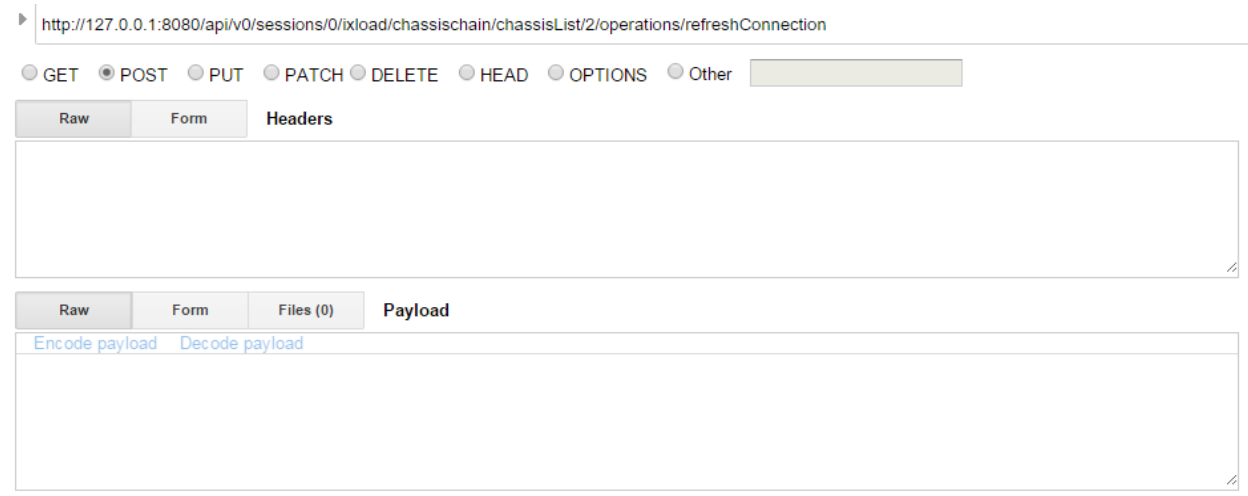


FIGURE 22 POST ON REFRESHCONNECTION

Status should be 202 Accepted as in Figure 23.

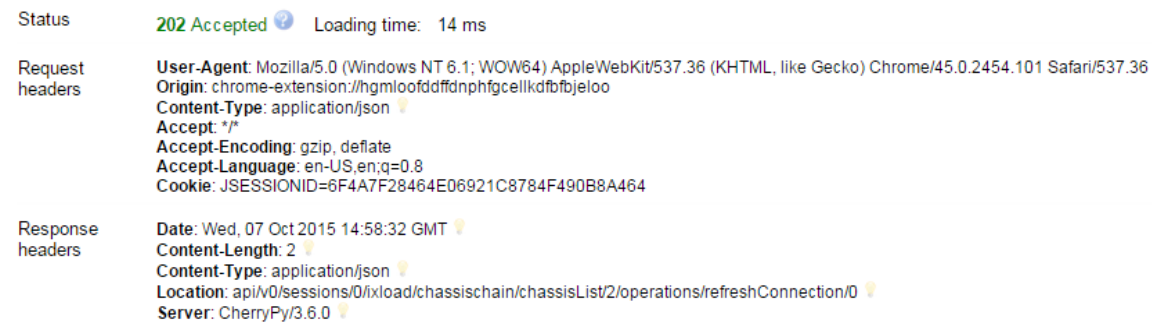


FIGURE 23 STATUS FOR REFRESHCONNECTION

The result to the refresh operation is posted below in Figure 24.

http://127.0.0.1:8080/api/v0/sessions/0/ixload/chassischain/chassisList/2/operations/refreshConnection/0

GET POST PUT PATCH DELETE HEAD OPTIONS Other

Raw Form Headers

Status **200 OK** Loading time: 16 ms

Request headers
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36
Content-Type: text/plain; charset=utf-8
Accept: */*
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Cookie: JSESSIONID=6F4A7F28464E06921C8784F490B8A464

Response headers
Date: Wed, 07 Oct 2015 15:03:18 GMT
Content-Length: 138
Content-Type: application/json
Server: CherryPy/3.6.0

Raw JSON Response

Copy to clipboard Save as file

```
{
  status: "Successful"
  actionName: "refreshConnection"
  state: "finished"
  refreshedChassis: "10.215.170.77"
}
```

FIGURE 24 RESULT TO REFRESHCONNECTION OPERATION

Status **200 OK** Loading time: 16 ms

Request headers
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36
Content-Type: text/plain; charset=utf-8
Accept: */*
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Cookie: JSESSIONID=6F4A7F28464E06921C8784F490B8A464

Response headers
Date: Wed, 07 Oct 2015 15:09:06 GMT
Content-Length: 297
Content-Type: application/json
Server: CherryPy/3.6.0

Raw JSON Response

Copy to clipboard Save as file

```
{
  status: "Successful"
  actionName: "refreshConnection"
  state: "finished"
  warning: "Could not connect to chassis 10.205.29.21. If any ports were assigned to the network they have been removed. Please reassign if chassis will be back up."
  refreshedChassis: ""
}
```

FIGURE 25 FAILED TO CONNECT

There is a new field inserted that is named *refreshedChassis*. This is referring to the ip or hostname of the chassis that was refreshed.

Usually this will contain the chassis that was refreshed. The only exception is when the loaded rxf has more than one chassis and all of them are not refreshed. In this case the *refreshedChassis* will hold all the chassis in the rxf because the whole chassis chain is getting refreshed.

There are cases when the rxf will have older chassis that no longer exist. To handle this in the refreshConnection operation a *warning* field was added that will instruct the user on what happened and the *refreshedChassis* field will contain only the chassis that were successfully connected to. In Figure 25 a GET on the status of refreshConnection operation will show that no chassis were refreshed and a warning message telling the user what happened.

Removing a chassis

This is done by performing a simple delete operation on the chassis list. In order to remove all the chassis in the list, the DELETE request must be performed on the chassis list url.

In order to remove only a certain chassis, the DELETE request must be performed on the following URL :

```
api/v0/sessions/0/ixload/chassischain/chassisList/chassisObjectId.
```

This is consistent with DELETE operations on other IxLoad Data Model lists.

Unassign ports

Unassigning ports is done by performing a DELETE request on the network port list. This is done the same as for removing chassis - the user can unassign either one of the ports (using the port object ID), or all the ports, by performing the DELETE operation on the list URL.

Assign ports

Assigning ports is done by performing POST operations on the network port list. The POST request must receive three parameters: chassisId, cardId, portId. These parameters do **not** represent the unique objectIDs used by REST API to identify resources as part of a list. These three parameters have the same meaning they have from UI and TCL/Python/Perl scripting, where a port is identified using a string such as "1.1.1".

The chassisId, cardId, portId can be seen on performing a GET request on the portList for each card of a chassis as shown in Figure 26.

```

http://127.0.0.1:8080/api/v0/sessions/0/ixload/chassischain/chassisList/1/cardList/0/portList
GET POST PUT PATCH DELETE HEAD OPTIONS Other
Raw Form Headers
Copy to clipboard Save as file
[13]
-0: {
  enableCapture: false
  portId: 1
  name: "Port 1.1.1"
  objectID: 26
  chassisId: 1
  managementIp: "10.0.1.1"
  -links: [1]
    -0: {
      href: "/api/v0/sessions/0/ixload/chassischain/chassisList/1/card
      rel: "portPersistentSetting"
    }
  analyzerPartialCapture: "False;20"
  cardType: "Xcellon-Ultra NP"
  cardId: 1
  id: "1.1.1"
  objectType: "ixPort"
}
-1: {
  enableCapture: false
  portId: 2
  name: "Port 1.1.2"
  objectID: 27

```

FIGURE 26 PORT LIST

The values highlighted above are the ones that will be used when assigning the port as in Figure 27.

```

http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/activeTest/communityList/0/network/portList
GET POST PUT PATCH DELETE HEAD OPTIONS Other
Raw Form Files (0) Payload
Encode payload Decode payload
{"chassisId":1, "cardId":1, "portId":1}

```

FIGURE 27 ASSIGN PORTS

IxVM Chassis (ixChassisBuilder)

You use the chassisBuilder object to configure and manage IxVM virtual chassis, and the cards and ports on them.

To get the root chassisBuilder object, send a GET request to the following URL:

```
http://serverAddress:8080/api/v0/sessions/{sessionId}/ixload/chassisBuilder
```

A response will be returned in the following form which indicates the connected chassis:

```
{
  "restObjectType": "ixChassisBuilder"
  "chassisName": "10.215.122.90"
  -"links": [1]
    -0: {
      "href": "/api/v0/sessions/0/ixload/chassisBuilder/docs"
      "rel": "docs"
    }
}
```

To display the list of operations available, send the following request:

```
http://serverAddress:8080/api/v0/sessions/{sessionId}/ixload/chassisBuilder/operations
```

```
{
  -"deleteCard": {
    "cardId": ""
  }
  -"updateChassisSettings": {
    "enableLicenseCheck": null
    "ntpServer": null
    "licenseServer": null
    "txDelay": null
  }
  "getChassisSettings": {}
  "hardChassisReboot": {}
  -"getCardPorts": {
    "cardId": ""
  }
  -"updatePortById": {
    "promiscMode": null
    "portId": ""
    "cardId": ""
    "lineSpeed": null
    "mtu": null
  }
  -"updateCard": {
    "cardServerId": ""
    "managementIp": null
    "keepAliveTimeout": null
  }
}
```

To execute an operation, send a POST request with the operation URL:

POST

```
http://serverAddress:8080/api/v0/sessions/{sessionId}/ixload/chassisBuilder/operations/getChassisSettings
```

You can retrieve operation's status by sending a GET with operation's ID:

GET

```
http://serverAddress:8080/api/v0/sessions/{sessionId}/ixload/chassisBuilder/operations/getChassisSettings/{operationId}
```

>

GET
 POST
 PUT
 PATCH
 DELETE
 HEAD
 OPTIONS
 Other

Raw Form Headers

Status: 200: OK ? Loading time:312ms

Response headers (4) Request headers (5)

Date: Mon, 21 Mar 2016 15:34:49 GMT
 Content-Length: 272
 Content-Type: application/json
 Server: CherryPy/3.6.0

Raw JSON Response

[COPY TO CLIPBOARD](#) [SAVE AS FILE](#)

```
{
  "status": "Successful"
  "actionName": "getChassisSettings"
  "state": "finished"
  "-links": [1]
    -0: {
      "href": "/api/v0/sessions/0/ixload/chassisBuilder/operations/getChassisSettings/0/result"
      "rel": "result"
    }
}
```

You can retrieve the operation's result by sending the following URL:

GET

```
http://serverAddress:8080/api/v0/sessions/{sessionId}/ixload/chassisBuilder/operations/getChassisSettings/{operationId}/ result }
```

The result will be specified in the links dict from the action status URL.

The result will be in the following form:

```
{
  "EnableLicenseCheck": 1
  "-links": [1]
    -0: {
      "href": "/api/v0/sessions/0/ixload/chassisBuilder/operations/getChassisSettings/0/result/docs"
      "rel": "docs"
    }
  "NtpServer": "10.215.170.157"
  "TxDelay": "1"
  "restObjectType": "ixChassisSettings"
  "LicenseServer": "10.215.122.90"
}
```

Statistics

The REST stat component will behave similar to the StatCollectorUtils component used in TCL. The user will be able to get available statistics for the activities configured in a test. He will also be able to apply filters on port/nettraffic/activity.

The user will be responsible for polling statistics from the web server. We will hold all statistics configured in the test in a circular buffer for a default amount of polls of 20 timestamps. As part for this release the number of default polls is not configurable.

Viewing statistics

Using IxLoad REST API users can configure L47 statistics that will be available at run time.

IxLoad REST API offers support for configuring and polling statistics for L47 protocols.

The root resource for stats in REST mode is found at the following URL:

- <http://127.0.0.1:8080/api/v0/sessions/0/ixload/stats>

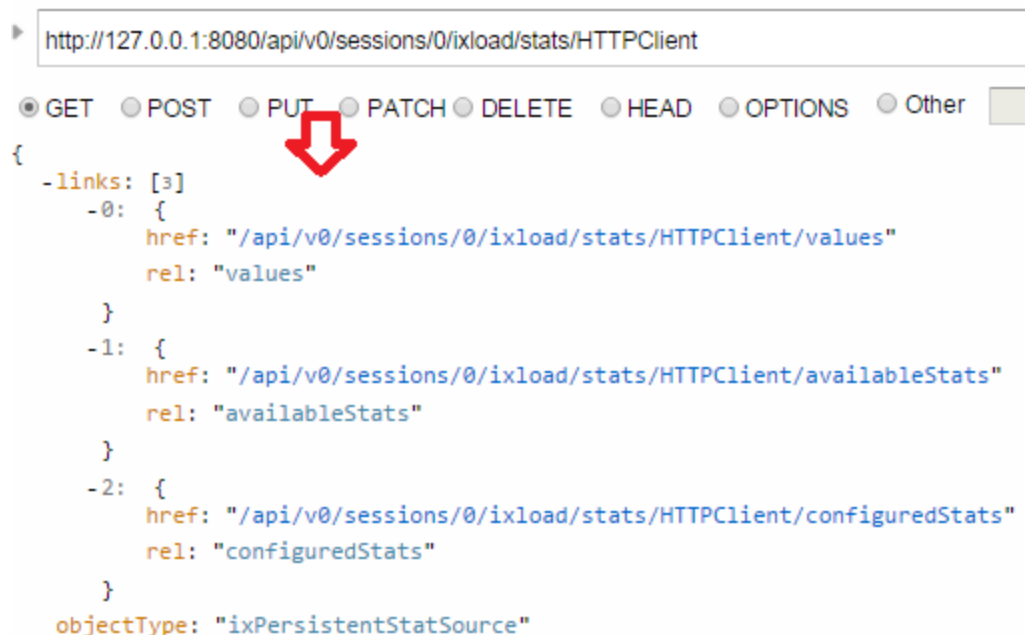
When a repository is loaded that contains L47 protocols, a GET request on this URL will return a list of stat sources, as seen below in Figure 28.



```
http://127.0.0.1:8080/api/v0/sessions/0/ixload/stats/
GET POST PUT PATCH DELETE HEAD OPTIONS Other
{
  -links: [4]
  -0: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPServerPerURL"
    rel: "HTTPServerPerURL"
  }
  -1: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPServer"
    rel: "HTTPServer"
  }
  -2: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPClientPerURL"
    rel: "HTTPClientPerURL"
  }
  -3: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPClient"
    rel: "HTTPClient"
  }
  objectType: "ixRestStatController"
}
```

FIGURE 28 STAT SOURCES

A GET request on any of the returned stat sources except RunState will return three lists: availableStats, configuredStats and values as below in Figure 29.



```

http://127.0.0.1:8080/api/v0/sessions/0/ixload/stats/HTTPClient

GET POST PUT PATCH DELETE HEAD OPTIONS Other

{
  -links: [3]
  -0: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPClient/values"
    rel: "values"
  }
  -1: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPClient/availableStats"
    rel: "availableStats"
  }
  -2: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPClient/configuredStats"
    rel: "configuredStats"
  }
  objectType: "ixPersistentStatSource"
}

```

FIGURE 29 GET STAT SOURCE

In Figure 29 there are several complex objects available that refer to how stats work in IxLoad REST framework:

- **availableStats:** this is the list that contains all available stats for the current test. This list is read-only - users can't remove available stats
- **configuredStats:** this is the list that contains the stats that have been configured for the current test. Here users can choose to enable / disable / remove / modify existing stats, by default configuredStats will include all availableStats
- Each configured stat resource has the following fields:
 - filterList
 - enabled
 - caption -> (this must be unique in the list)
 - objectID -> (also must be unique)
 - aggregationType
 - statName
- **values :** this is a dictionary that will contain the actual stat values during the IxLoad test run
 - If a GET request is performed on 'values' before the test actually runs, an empty dictionary will be returned

- The format for the dictionary will be: {timestamp : { stat name : stat value }} The 'values' dictionary will only keep, at all times, the last 20 timestamps. If the user does not poll stats in due time, he might lose some timestamps.

Figure 30 shows the values obtained when running a query on the HTTP Client stat values.

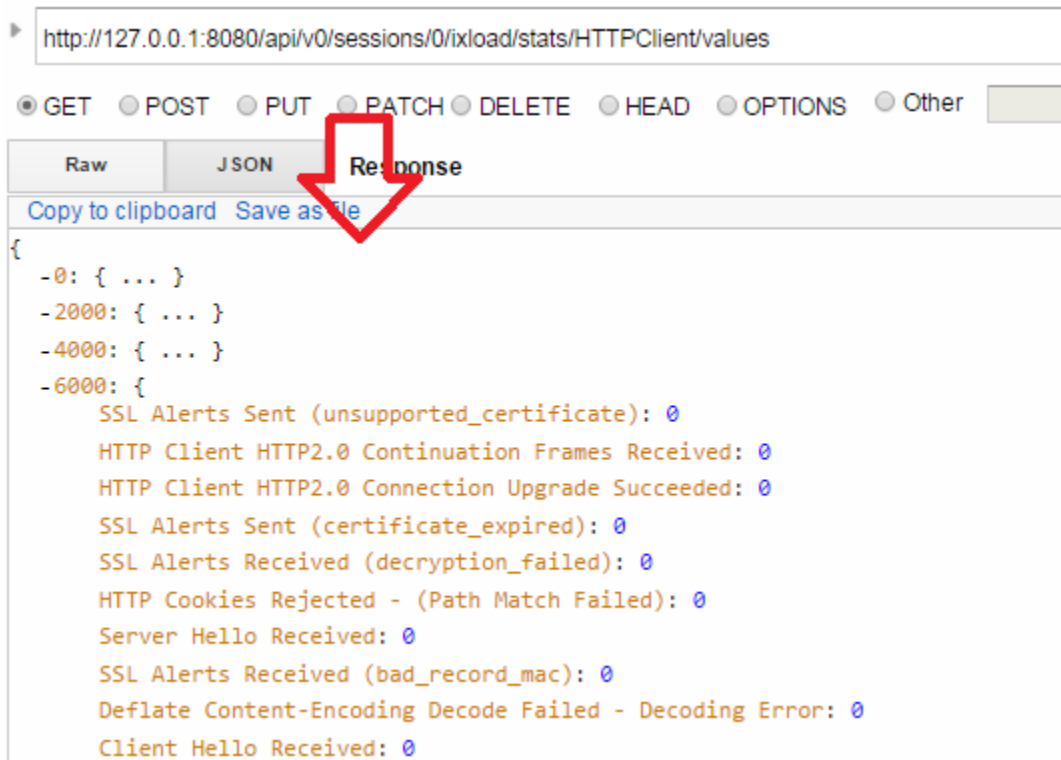


FIGURE 30 STAT VALUES

RunState Stat Source

The RunState stat source is listed for all agents under a single stat source called 'RunState'. There are no configurable options for the RunState stat source, you can only perform 'GET' requests on it. The only option for RunState is the 'values' option; it does not have the 'availableStats' or 'configuredStats' options.

The URL for the RunState stat source is:

<http://IP:8080/api/v0/sessions/sessionId/ixload/stats/RunState>

which simply contains a link to the 'values' resource. The stat values can be viewed at the following URL:

<http://IP:8080/api/v0/sessions/sessionId/ixload/stats/RunState/values>

A 'GET' on the values URL before the test starts running will return an empty dictionary. After the test starts running, the dictionary will be populated with the RunState stat values for all agents.

Modifying configured statistics

Changing statistics is done by running the PATCH method on the configured statistics structure.

Statistics can be enabled or disabled and the aggregation type can also be changed.

Figure 31 shows the url for getting a configured stat.

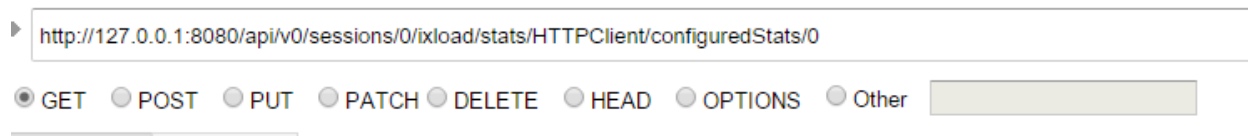


FIGURE 31 GET CONFIGURED STATS

The result of the get in the above request is shown in Figure 32.



FIGURE 32 GET CONFIGURED STATS DATA

To change a configured stat a PATCH method will be issued like in Figure 33. The payload must contain the properties that require to be changed.

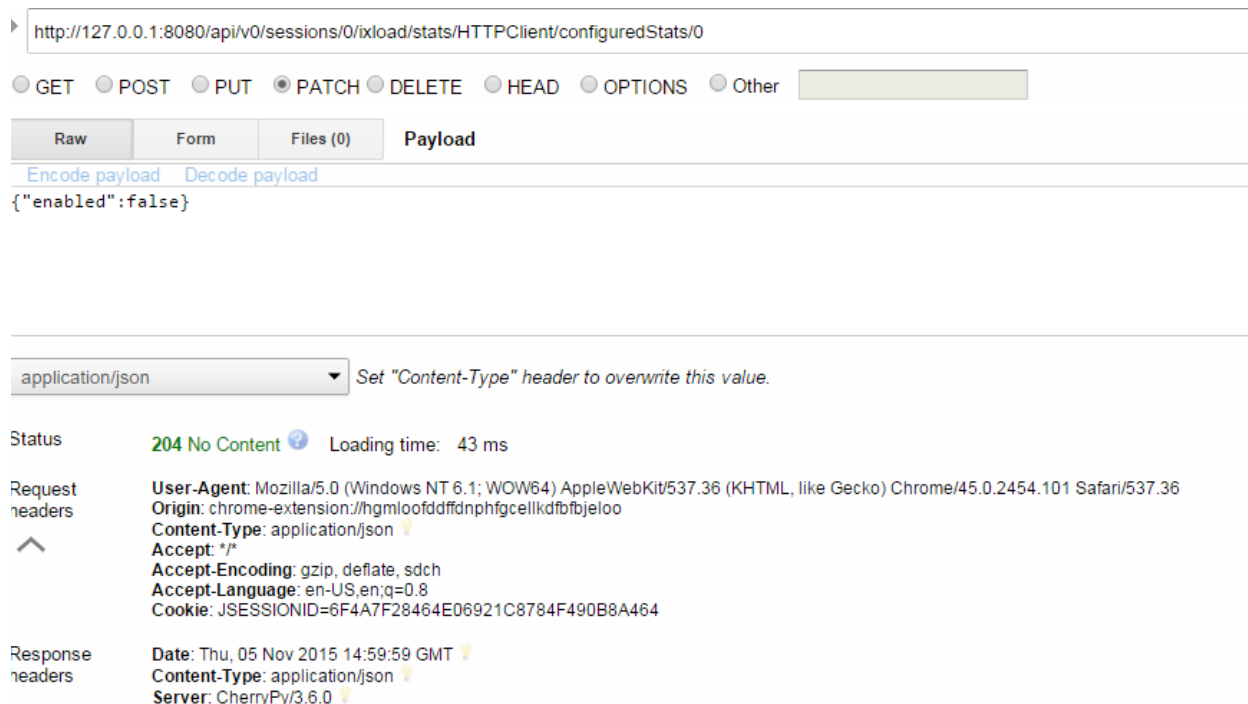
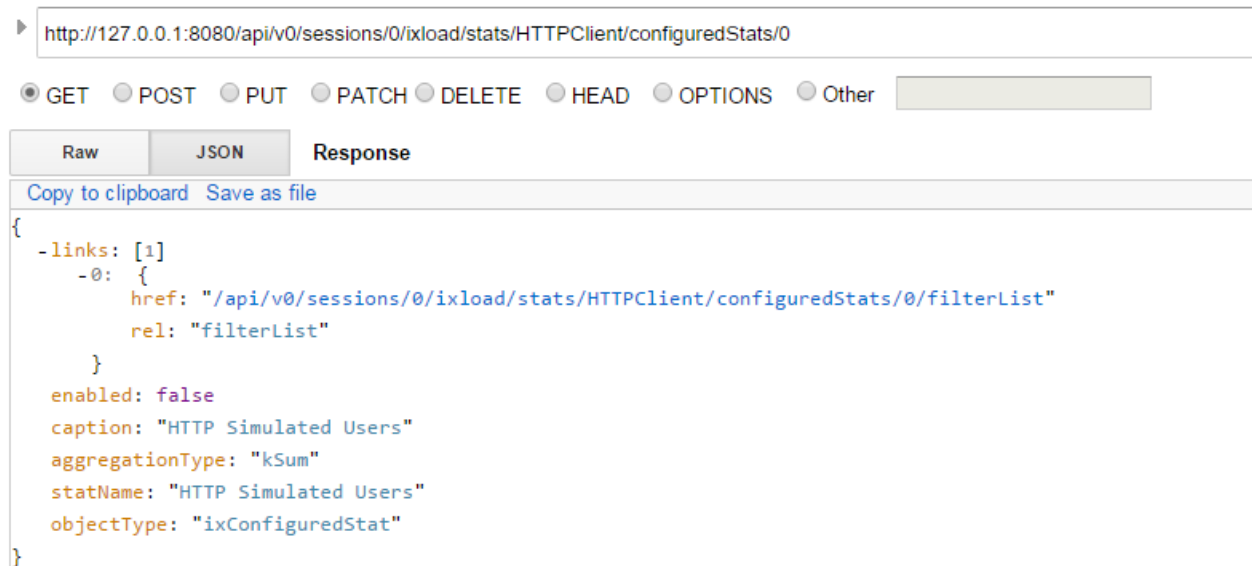


FIGURE 33 PATCH ON CONFIGURED STATS

Figure 34 shows how the PATCH method above changed the configured stat structure by disabling it.



The screenshot shows a REST client interface with the following elements:

- URL: `http://127.0.0.1:8080/api/v0/sessions/0/ixload/stats/HTTPClient/configuredStats/0`
- Method: GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS, Other
- Response format: Raw, JSON, Response
- Buttons: [Copy to clipboard](#), [Save as file](#)
- JSON Response:

```
{
  -links: [1]
  -0: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPClient/configuredStats/0/filterList"
    rel: "filterList"
  }
  enabled: false
  caption: "HTTP Simulated Users"
  aggregationType: "kSum"
  statName: "HTTP Simulated Users"
  objectType: "ixConfiguredStat"
}
```

FIGURE 34 CONFIGURED STATS AFTER PATCH

Filtering stats

The filter stats are accessed like in [Figure 35 Configured stats – filter List](#) by issuing a GET command on the filterlist from a specific configuredStat item.

The screenshot shows a REST client interface. At the top, the URL is `http://127.0.0.1:8080/api/v0/sessions/0/ixload/stats/HTTPClient/configuredStats/0/filterList`. Below the URL, there are radio buttons for HTTP methods: GET (selected), POST, PUT, PATCH, DELETE, HEAD, OPTIONS, and Other. There are also tabs for 'Raw', 'Form', and 'Headers'. Below this, there are tabs for 'Raw', 'JSON', and 'Response'. The 'Response' tab is active, showing a JSON response. Above the JSON, there are links for 'Copy to clipboard' and 'Save as file'. The JSON response is as follows:

```
{
  -links: [4]
  -0: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPClient/configuredstats/0/filterlist/cardFilters"
    rel: "cardFilters"
  }
  -1: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPClient/configuredstats/0/filterlist/activityFilters"
    rel: "activityFilters"
  }
  -2: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPClient/configuredstats/0/filterlist/chassisFilters"
    rel: "chassisFilters"
  }
  -3: {
    href: "/api/v0/sessions/0/ixload/stats/HTTPClient/configuredstats/0/filterlist/portFilters"
    rel: "portFilters"
  }
  objectType: "ixRestFilters"
}
```

FIGURE 35 CONFIGURED STATS – FILTER LIST

A configured stat contains several filters each enabling the option to get the values at:

- Card level
- Activity level
- Chassis level
- Port level

Adding a port filter is demonstrated in Figure 36 by adding a new port to the portFilter list.

▶

GET
 POST
 PUT
 PATCH
 DELETE
 HEAD
 OPTIONS
 Other

[Encode payload](#)
[Decode payload](#)

```

{"value": "10.215.170.45/Card1/Port1"}
    
```

Set "Content-Type" header to overwrite this value.

Status 201 Created Loading time: 47 ms

Request headers

```

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36
Origin: chrome-extension://hgmlfoofddfdnphfgcellkdfbfjeloo
Content-Type: application/json
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8
Cookie: JSESSIONID=6F4A7F28464E06921C8784F490B8A464
    
```

Response headers

```

Date: Thu, 05 Nov 2015 15:04:12 GMT
Content-Length: 2
Content-Type: application/json
Location: /api/v0/sessions/0/ixload/stats/HTTPClient/configuredStats/0/filterList/portFilters/0
Server: CherryPy/3.6.0
    
```

FIGURE 36 POST ON PORT FILTERS

Figure 37 shows how the filter looks after it was added above.

▶

GET
 POST
 PUT
 PATCH
 DELETE
 HEAD
 OPTIONS
 Other

[Copy to clipboard](#)
[Save as file](#)

```

[1]
  -0: {
    objectID: 0
    value: "10.215.170.45/Card1/Port1"
    objectType: "ixRestFilter"
  }
    
```

FIGURE 37 GET PORT FILTER

Several filters can be set for multiple configured stats according to how the user will need to see the statistics. Aggregations and processing can be done in the client script once the stats are coming in.

Adding an Activity Filter

Adding an activity filter to a statistic can be done by executing a POST request on an URL similar to:

<http://127.0.0.1:8080/api/v0/sessions/0/ixload/stats/HTTPClient/configuredStats/0/filterList/activityFilters>

with the following payload:

```
{"value": "Traffic1@Network1 - HTTPClient1"}
```

where Traffic1@Network1 is the nettraffic name (formed by the traffic and the network name) and HTTPClient1 is the activity name

Generated CSVs

During the IxLoad test run, CSVs files will also be generated. If the user does not change any settings regarding the CSV path, they will be generated in the default result directory which can be configured in IxLoad UI.

If the user wants to save the generated CSVs in a custom path, he must set the following two variables on the 'test' resource, before running the configuration:

- PATCH on <http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/>
- payload:
 - outputir : true (the default is 'false')
 - runResultDirFull : "F:\\path\\to\\the\\new\\result\\dir"

Logging

You can retrieve log messages by accessing a URL. The logs available from the REST API will be equivalent to the entries seen in the IxLoad UI. The URL where log entries are accessible is the following:

- <http://127.0.0.1:8080/api/v0/sessions/0/ixload/test/logs>

A GET applied to the logs URL will return a list of the last log entries. By default, the last 100 entries will be shown, but this number can be changed from the 'preferences' URL. Each log entry will contain the moduleName, severity, timestamp and message:

```
[
  {
    "module": "ixChassisChain",
    "severity": "Info",
    "objectID": 3,
    "timeStamp": "2016/06/02 19:03:35.838",
    "links": [
      {
        "href": "/api/v0/sessions/0/ixload/test/logs/3/docs",
        "rel": "docs"
      }
    ],
    "restObjectType": "ixRestLogEntry",
    "message": "Validating that 10.215.122.22 accepts incoming connections. Will try to connect for 10 seconds."
  },
  {
    "module": "ixChassisChain",
    "severity": "Info",
    "objectID": 4,
    "timeStamp": "2016/06/02 19:03:36.852",
    "links": [
      {
        "href": "/api/v0/sessions/0/ixload/test/logs/4/docs",
        "rel": "docs"
      }
    ],
    "restObjectType": "ixRestLogEntry",
    "message": "Validation for 10.215.122.22 is completed. IP is valid and connection succeeded."
  }
]
```

FIGURE 38 LOGS

REST Script Templates

Scripts that create and use an IxLoad session in REST mode can be written in any language that supports performing REST requests. An installed IxLoad build contains a set of sample scripts that perform basic IxLoad operations from REST using python. The samples can be found in root installation folder of IxLoad in a subfolder named RestScripts.

The sample folder contains 4 python scripts. The 4 scripts located in the samples folder need to be run with a Python executable that has the 'requests' module installed, as it is described in the README.txt file included in the RestScripts folder. The scripts do not require any command line arguments. They can be simply executed by performing 'python.exe SimpleRun.py'.

Before running the scripts please change the configuration data (IxLoad Version, chassis, rxf file path) in the beginning of each script accordingly.

The provided templates act as guidance, the IxLoad REST API is compatible with any programming language that supports running HTTP requests.

The REST script templates are relying on 2 utility python files:

- IxLoadUtils.py which deals with specific IxLoad REST API functionality
- IxLoadRestUtils.py which deals with providing the underlying abstraction level that IxLoadUtils is using to receive, interpret and dispatch requests.

The two files are helpers to implement a Python script to handle REST communication with the IxLoad REST framework. The template files expose basic workflow scenarios as examples for users to understand how to automatically configure IxLoad through REST.

AddNewCommand

This template will:

- Create a session
- Load an Rxf
- Clear the chassis list
- Add a chassis
- Assign ports to the networks
- Clear the command list for client activity
- Update the command list of the client HTTP activity by:
 - o Adding a GET command with custom properties
 - o Adding a POST command with custom properties
- Save the Rxf
- Start the test
- Poll the stats
- Close the IxLoad session

ChangeAgentObjectives

This template will:

- Create a session
- Load an Rxf
- Clear the chassis list
- Add a chassis
- Assign ports to the networks
- Update the activity options by:
 - o Enabling constraints
 - o Setting a constraint value
 - o Changing the objective type
 - o Setting a new objective type
- Save the Rxf
- Start the test
- Poll the stats
- Close the IxLoad session

ChangeIpType

This template will:

- Create a session
- Load an Rxf
- Clear the chassis list
- Add a chassis
- Assign ports to the networks
- Update the ip Ranges changing the count and the ipAddress
- Save the Rxf
- Start the test

- Poll the stats
- Close the IxLoad session

SimpleRun

This template will:

- Create a session
- Load an Rxf
- Clear the chassis list
- Add a chassis
- Assign ports to the networks
- Save the Rxf
- Start the test
- Poll the stats
- Close the IxLoad session

IxLoadRestUtils

This module defines the following

```
class Connection(__builtin__.object)
```

This is the class that executes the HTTP requests to the application instance. It handles creating the HTTP session and executing HTTP methods.

Methods defined here:

- `__init__(self, siteUrl, apiVersion)`

Args:

- o `siteUrl` is the actual url to which the Connection instance will be made.
- o `apiVersion` is the actual version of the REST API that the Connection instance will use.
- o The HTTP session will be created when the first http request is made.
- `httpDelete(self, url="", data="", params={}, headers={})`
Method for calling HTTP DELETE. Will return the HTTP reply.
- `httpGet(self, url="", data="", params={}, headers={})`
Method for calling HTTP GET. This will return a WebObject that has the fields returned in JSON format by the GET operation.
- `httpPatch(self, url="", data="", params={}, headers={})`
Method for calling HTTP PATCH. Will return the HTTP reply.
- `httpPost(self, url="", data="", params={}, headers={})`
Method for calling HTTP POST. Will return the HTTP reply.
- `httpRequest(self, method, url="", data="", params={}, headers={})`

Args:

- o Method (mandatory) represents the HTTP method that will be executed.
- o url (optional) is the url that will be appended to the application url.
- o data (optional) is the data that needs to be sent along with the HTTP method as the JSON payload
- o params (optional) the payload python dict not necessary if data is used.
- o headers (optional) these are the HTTP headers that will be sent along with the request. If left blank will use default

Method for making a HTTP request. The method type (GET, POST, PATCH, DELETE) will be sent as a parameter. Along with the url and request data. The HTTP response is returned

Class methods defined here:

- `urljoin(cls, base, end)` from `__builtin__.type`

Join two URLs. If the second URL is absolute, the base is ignored. Use this instead of `urlparse.urljoin` directly so that we can customize its behavior if necessary.

Currently differs in that it

1. Appends a / to base if not present.
2. Casts end to a str as a convenience

Data descriptors defined here:

- `__dict__`
dictionary for instance variables (if defined)
- `__weakref__`
list of weak references to the object (if defined)

Data and other attributes defined here:

- `kContentType = 'application/json'`
- `kHeaderContentType = 'content-type'`

```
class WebList(__builtin__.list)
```

By using this class a JSON list will be transformed in a list of `WebObject` instances.

Methods defined here:

- `__init__(self, entries=[])`

Create a `WebList` from a list of items that are processed by the `_WebObject` function

Data descriptors defined here:

- `__dict__`
dictionary for instance variables (if defined)

- `__weakref__`
list of weak references to the object (if defined)

```
class WebObject(__builtin__.object)
```

A WebObject instance will have its fields set to correspond to the JSON format received on a GET request. For example: a response in the format: {"caption": "http"} will return an object that has obj.caption="http"

Methods defined here:

- `__init__(self, **entries)`
Create a WebObject instance by providing a dict having a property - value structure.
- `getOptions(self)`
Get the JSON dictionary which represents the WebObject Instance

Data descriptors defined here:

- `__dict__`
dictionary for instance variables (if defined)
- `__weakref__`
list of weak references to the object (if defined)

Functions

- `formatDictToJSONPayload(dictionary)`
Converts a given python dict instance to a string JSON payload that can be sent to a REST API.
- `getConnection(server, port)`
Gets a Connection instance, which will be used to make the HTTP requests to the application

IxLoadUtils

The IxLoadUtils module is a collection of specific functions that deal with common IxLoad workflows.

Functions

- `addChassisList(connection, sessionUrl, chassisList)`
This method is used to add one or more chassis to the chassis list.
Args:

- o connection is the connection object that manages the HTTP data transfers between the client and the REST API
- o sessionUrl is the address of the session that should run the test
- o chassisList is the list of chassis that will be added to the chassis chain.
- addCommands(connection, sessionUrl, commandDict)

This method is used to add commands to a certain list of provided agents.

Args:

- o connection is the connection object that manages the HTTP data transfers between the client and the REST API
- o sessionUrl is the address of the session that should run the test
- o commandDict is the Python dict that holds the mapping between agent name and specific commands. (commandDict format -> { agent name : [{ field : value }] })
- assignPorts(connection, sessionUrl, portListPerCommunity)

This method is used to assign ports from a connected chassis to the required NetTraffics.

Args:

- o connection is the connection object that manages the HTTP data transfers between the client and the REST API
- o sessionUrl is the address of the session that should run the test
- o portListPerCommunity is the dictionary mapping NetTraffics to ports (format -> { community name : [port list] })
- changeActivityOptions(connection, sessionUrl, activityOptionsToChange)

This method will change certain properties for the provided activities.

Args:

- o connection is the connection object that manages the HTTP data transfers between the client and the REST API
- o sessionUrl is the address of the session that should run the test
- o activityOptionsToChange is the Python dict that holds the mapping between agent name and specific properties (activityOptionsToChange format: { activityName : { option : value } })
- changeCardsInterfaceMode (connection, chassisChainUrl, chassisIp, cardIdList, mode)

This method is used to change the interface mode on a list of cards from a chassis. In order to call this method, the desired chassis must be already added and connected.

- o connection is the connection object that manages the HTTP data transfers between the client and the REST API
- o chassisChainUrl is the address of the chassisChain resource
- o chassisIp is the IP or hostname of the chassis that contains the card(s)
- o cardIdList is a list of card IDs
- o mode is the interface mode that will be set on the cards. Possible options are (depending on card type): 1G, 10G, 40G, 100G
- changeIpRangesParams(connection, sessionUrl, ipOptionsToChangeDict)

This method is used to change certain properties on an IP Range.

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- sessionUrl is the address of the session that should run the test
- ipOptionsToChangeDict is the Python dict holding the items in the IP range that will be changed.
- (ipOptionsToChangeDict format -> { IP Range name : { optionName : optionValue } })

- clearAgentsCommandList(connection, sessionUrl, agentNameList)

This method clears all commands from the command list of the agent names provided.

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- sessionUrl is the address of the session that should run the test
- agentNameList the list of agent names for which the command list will be cleared.

- clearChassisList(connection, sessionUrl)

This method is used to clear the chassis list. After execution no chassis should be available in the chassisList.

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- sessionUrl is the address of the session that should run the test

- createSession(connection, ixLoadVersion)

This method is used to create a new session. It will return the url of the newly created session

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- ixLoadVersion this is the actual IxLoad Version to start

- deleteSession(connection, sessionUrl)

This method is used to delete an existing session.

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- sessionUrl is the address of the session to delete

- `getCommandListUrlForAgentName(connection, sessionUrl, agentName)`

This method is used to get the `commandList` url for a provided agent name.

Args:

- o `connection` is the connection object that manages the HTTP data transfers between the client and the REST API
- o `sessionUrl` is the address of the session that should run the test
- o `agentName` is the agent name for which the `commandList` address is provided

- `getIPRangeListUrlForNetworkObj(connection, networkUrl)`

This method will return the IP Ranges associated with an IxLoad Network component.

Args:

- o `connection` is the connection object that manages the HTTP data transfers between the client and the REST API
- o `networkUrl` is the REST address of the network object for which the network ranges will be provided.

- `getTestCurrentState(connection, sessionUrl)`

This method gets the test current state. (for example - running, unconfigured, ..)

Args:

- o `connection` is the connection object that manages the HTTP data transfers between the client and the REST API
- o `sessionUrl` is the address of the session that should run the test.

- `getTestRunError(connection, sessionUrl)`

This method gets the error that appeared during the last test run.

If no error appeared (the test ran successfully), the return value will be 'None'.

Args:

- o `connection` is the connection object that manages the HTTP data transfers between the client and the REST API.
- o `sessionUrl` is the address of the session that should run the test.

- `loadRepository(connection, sessionUrl, rxfFilePath)`

This method will perform a POST request to load a repository.

Args:

- o `connection` is the connection object that manages the HTTP data transfers between the client and the REST API

- sessionUrl is the address of the session to load the rxf for
- rxfFilePath is the local rxf path on the machine that holds the IxLoad instance

- performGenericDelete(connection, listUrl, payloadDict)

This will perform a generic DELETE method on a given url

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- url is the address of where the operation will be performed
- payloadDict is the python dict with the parameters for the operation

- performGenericOperation(connection, url, payloadDict)

This will perform a generic operation on the given url, it will wait for it to finish.

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- url is the address of where the operation will be performed
- payloadDict is the python dict with the parameters for the operation

- performGenericPatch(connection, url, payloadDict)

This will perform a generic PATCH method on a given url

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- url is the address of where the operation will be performed
- payloadDict is the python dict with the parameters for the operation

- performGenericPost(connection, listUrl, payloadDict)

This will perform a generic POST method on a given url

Args:

- connection is the connection object
- url is the address of where the operation will be performed
- payloadDict is the python dict with the parameters for the operation

- pollStats(connection, sessionUrl, watchedStatsDict, pollingInterval=4)

This method is used to poll the stats. Polling stats is per request but this method does a continuous poll.

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- sessionUrl is the address of the session that should run the test
- watchedStatsDict these are the stats that are being monitored
- pollingInterval the polling interval is 4 by default but can be overridden.

- runTest(connection, sessionUrl)

This method is used to start the currently loaded test. After starting the 'Start Test' action, wait for the action to complete.

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- sessionUrl is the address of the session that should run the test.

- saveRxf(connection, sessionUrl, rxfFilePath)

This method saves the current rxf to the disk of the machine on which the IxLoad instance is running.

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- sessionUrl is the address of the session to save the rxf for
- rxfFilePath is the location where to save the rxf on the machine that holds the IxLoad instance

- setCardsAggregationMode(connection, chassisChainUrl, chassisIp, cardIdList, mode)

This method is used to change the aggregation mode on a list of cards from a chassis. In order to call this method, the desired chassis must be already added and connected

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API

- chassisChainUrl is the address of the chassisChain resource
- chassisIp is the IP or hostname of the chassis that contains the card(s)
- cardIdList is a list of card IDs
- mode is the aggregation mode that will be set on the cards. Possible options are (depending on card type): NA (Non Aggregated), 1G, 10G, 40G

- waitForActionToFinish(connection, replyObj, actionUrl)

This method waits for an action to finish executing. After a POST request is sent in order to start an action,

The HTTP reply will contain, in the header, a 'location' field, that contains an URL.

The action URL contains the status of the action. This will perform a GET on that URL every 0.5 seconds until the action finishes with a success.

If the action fails, this will throw an error and print the action's error message.

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- replyObj the reply object holding the location
- actionUrl - the url pointing to the operation

`waitForAllCaptureData(connection, sessionUrl)`

This method is used to wait for the test to capture all the port data that was received after the test has finished running.

Args:

- connection is the connection object that manages the HTTP data transfers between the client and the REST API
- sessionUrl is the address of the session that should run the test

The IxLoad REST templates provide a basic way of doing common tasks with IxLoad. The same can be achieved in other programming languages, not necessarily python.