Infoblox
CONTROL YOUR NETWORK

Infoblox Network Automation API Guide

API Version 3.0

Copyright Statements

Trademark Statements

Company Information

Web: http://www.infoblox.com/

Product Information

Document Number: 400-0373-005

Document Updated: July 14, 2015

Warranty Information

Your purchase includes a 90-day software warranty and a one year limited warranty on the Infoblox appliance, plus an Infoblox Warranty Support Plan and Technical Support. For more information about Infoblox Warranty information, refer to the Infoblox Web site, or contact Infoblox Technical Support.

Infoblox
CONTROL YOUR NETWORK

Infoblox
CONTROL YOUR NETWORK

Network Automation API Developer's Guide

Introduction

Network Automation collects, organizes and displays information in an array of customizable tables, graphs and reports covering virtually every aspect of network operations. Network Automation search functions can be used to find and filter stored information. Many of Network Automation's information and results displays can be exported as CSV files, Excel spreadsheets and PDF documents. Before developing a program to extract data from the Network Automation database using the API, determine whether it would be easier to obtain the needed information through the Network Automation GUI.

The Network Automation Application Programming Interface (API) enables external programs to retrieve information about devices, interfaces, VLANs and other network entities from the Network Automation database, and to retrieve information about neighbor relationships between devices, and send commands to Network Automation.

The API is intended for use by customers wanting to leverage or reformat information from the Network Automation database for use in other system and applications. Typical applications include:

- Custom dashboards. External programs can extract data from Network Automation to populate portions of dashboards and displays that integrate data from multiple systems. Such applications may consolidate status, trends and urgent issues in operations centers or management displays.
- Custom reports. External programs may transfer Network Automation data to third-party reporting systems to create reports with specialized content and with standard formatting. These reports may combine data from multiple systems.
- CMDB. Configuration Management Database will leverage Network Automation as the trusted source for network device asset inventory, properties, topology, configuration and health.
- Asset management systems. Some organizations have separate asset management systems and can obtain asset information from Network Automation, similar to CMDB projects.

Primary changes in the 3.0 API consist of updates to existing models, objects and methods to include the necessary relationships with newly added nested device group objects.

Note: The prior Release 2.10 API, replaces the previous .Networks. API with the generalized .VirtualNetwork. API. Avoid using the .Networks. API for any applications.

The language-independent API Core Version 3.0 provides bidirectional functionality, enabling clients to both retrieve data and send commands to Network Automation. This enables clients to:

- Send HTTP requests using GET, or using POST with MIME types application/x-www-form-urlencoded, multipart/form-data, application/json, or application/xml.
- Receive HTTP responses with MIME type application/json or application/xml.
- Query and search over 70 different types of objects within the Network Automation data model.

- Configure the Network Automation discovery ranges and other settings.
- Schedule and run jobs on the Network Automation programmatically.
- Associate job custom fields with jobs programmatically.

  See the following chapter, *Network Automation API Core*, for more information on usage of and migration to the API Core Version 3.0.
- The Perl API is a set of Perl modules built on top of the API Core. These modules handle authentication, formatting of URL requests and parsing of XML returned documents. This implementation of the API enables more focus on program logic. Network Automation includes a Perl package providing adherence to Perl conventions, and added functionality. See *Migration to API Core Version 3.0* for more information.

The system maintains security by requiring authentication based on the same user names and passwords used by the Network Automation graphical user interface.

Effective use of the API requires programming skills. If you are not a programmer, you will need a programmer's help to develop applications to query the Network Automation database through the API, process the resulting data and send commands. While any language can be used with the API Core, Perl is an appropriate choice. As its name implies, interactions with the Perl API are implemented using Perl scripts.

Note: Infoblox may release future versions of the APl that are not backward compatible with the current version.

Network Automation API Core

API Versioning

Specific versions of the API are available at `<your Network Automation URI>/api/X`, where X is a version string for the API version.

This chapter discusses API Core 3.0.

To maintain version flexibility, a client should determine the latest version root using the `/api/base_uri` call, which is independent of the version and does not require authentication. This API call accepts a version string, and returns the URI root that the client should use for that version of the API.

The Network Automation Perl API package automatically determines the available API version. If you upgrade Network Automation, but do not upgrade the Network Automation API module, the client host will continue to access the older API. This prevents scripts using the API from breaking immediately, although Infoblox strongly recommends upgrades as soon as possible to the latest API version to assure continued forward compatibility.

After authenticating, the client may also call `/api/server_info`, which will also always be available at that un-versioned URI. This method returns a variety of version information, including the most recent version of the API supported by the server.

Backward Compatibility

API versioning is provided as a convenience to client authors in order to allow additional time to adapt to an updated API. It is not intended to keep all API calls accessible for all time. The following limitations are placed on versioning:

- In general, data and object model changes will **not** maintain backward compatibility, in the sense that data will not be converted to the "old" view when being returned from an older API call version. However, following the best practices will minimize any disruption in client implementations:

For client authors:

- Do not rely on any fields existing in any given data object, except for those that are explicitly needed for client functionality. For example, do not build compiled objects that require all fields to be available in the XML or JSON data fields.
- Do not rely on the same number of fields returning for a data object; additional fields may be added in future versions.
- Do not rely on the ordering of fields in the XML or JSON output. Ordering is significant for "array" objects, but field ordering for data objects or key ordering for hash objects is not deterministic.
- API calls will be maintained for at least one "major API version," subject to the following:
  - "Major API version" is independent of the overall product version.

- No more than one major API version will be released within any twelve month period, although there may be exceptions to this if important changes are needed.
- Any API change that affects backward compatibility will be well documented and typically published for at least one year before release.

Network Automation API Protocol

The underlying protocol used for the Network Automation API is HTTP /1.1, which is described in RFC2616, http://www.w3.org/Protocols/rfc2616/rfc2616.html.

This section provides more detail describing how the API builds on top of HTTP to allow access and control of Network Automation. For purposes of this document, HTTP and HTTPS are the same. That is, the interface supports both http and https URL schemes; whether on a plain or SSL socket, the API will work in the same manner.

This section is especially relevant if you intend to access the Core API directly; it is not necessary for users of the Perl API to understand all the details here.

Request Format

The HTTP standard requires the requests to consist of a request line, headers, and an optional request body. The Network Automation does not utilize any special request methods or headers beyond the HTTP/1.1 standard.

For POST operations, the client must send a request body in addition to the request line and headers. Network Automation supports several different request body content-types: application/x-www-form-urlencoded, multipart/form-data, application/json, application/xml. Most calls will accept any of these formats as equivalent. However, certain calls may specify that they accept only a subset of these content-types.

application/x-www-form-urlencoded, multipart/form-data

These two content-types can be described as “form encodings”, and are commonly used by browsers to POST form data.

The application/x-www-form-urlencoded content-type is the default content-type that browsers use when POSTing a form and is defined in the HTML specification. It encodes the parameters in the same manner as a GET request, but places them in the request body rather than the request line. This allows much more data to be sent with the request than in a GET, but is still not sufficient for file uploads.

The multipart/form-data content-type fills this gap by encoding data in a way that allows more complex data types. It is defined in RFC 2388 (http://tools.ietf.org/html/rfc2388), but most user-agent libraries will support it natively.

application/json, application/xml

The application/json content-type refers to the JavaScript Object Notation (JSON), as described in RFC 4627 (http://tools.ietf.org/html/rfc4627). The application/xml content-type, or the now deprecated text/xml content-type, is used to identify Extensible Markup Language (XML) in the request body. Network Automation supports XML 1.0 as defined in the W3C standard (http://www.w3.org/TR/2008/REC-xml-20081126/).

	JSON	XML

Format	{
	<name>: <value>
	<repeat as necessary, delimited by commas>
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<request>
		<name>value</name>
		Repeat as necessary
		</request>

	JSON	XML

Example	<pre>{
	id: 45,
	"user_name": "jsmith",
	"first_name": "John",
	"last_name": "Smith"
	}</pre>
		<pre><?xml version="1.0"
		encoding="UTF-8"?>
		<request>
		<id>45</id>
		<user_name>jsmith</user_name>
		<first_name>John</first_name>
		<last_name>Smith</last_name>
		</request></pre>

Unless otherwise defined by the specific API call documentation, JSON requests should consist of an anonymous hash of name/value pairs, whereas XML requests should consist of a request root element with child elements representing the name/value parameter pairs.

Some API calls use more complex JSON or XML input formats. Some functionality of these calls may only accessible through a POST with one of these content-types, rather than the form encodings.

Response Format

The HTTP 1.1 standard dictates that the responses consist of a status line, headers, and a response body. In addition, the Network Automation API has standardized response bodies for some of the specific status codes that may be returned.

Status Codes

The Network Automation API utilizes standard HTTP 1.1 status codes as defined in RFC2616, Section 6. One additional standard code, 102 Processing from RFC2518, is also used.

One non-standard code, 265 Deprecated, is used to indicate that the API call used is a deprecated call. This should be treated as a 200 OK with regard to processing, but provides a warning that the call may go away in a future release.

The status codes below are commonly used by Network Automation. Other codes may be returned based on standard HTTP semantics.

Code	Reason Phrase	Description
102	Processing	This code indicates that the server has received and is processing the request, but no response is available yet.
200	OK	The request succeeded. Standard HTTP 1.1 status code.
201	Created	The request to create an entity succeeded. Standard HTTP 1.1 status code.
202	Accepted	The request has been accepted for processing, but the processing has not been completed. Standard HTTP 1.1 status code.
265	Deprecated	The request succeeded, and does not need to be repeated. However, the specified API call is deprecated and will be removed in a future version.
301	Moved Permanently	The requested resource is assigned a new permanent URI. Standard HTTP 1.1 status code.
400	Bad Request	Used for application-level error responses. For example, this will be used when a form submission fails to pass the form validations.
403	Forbidden	The server understood the request, but is refusing to fulfill it. Standard HTTP 1.1 status code, but the Network Automation adds a payload to determine if this is due to access controls or lack of authentication (see below).
404	Not Found	The server has not found anything matching the Request-URI. Standard HTTP 1.1 status code.

Code	Reason Phrase	Description
409	Conflict	The server cannot process the request due to a conflict. This generally means that an attempt to create a new resource violates a uniqueness constraint.
410	Gone	The requested resource is no longer available at the server and no forwarding address is known. Standard HTTP 1.1 status code, used when a subsequent version obsoletes an API call.
413	Request Entity Too Large	The server is refusing to process a request because the request entity is larger than the server is willing or able to process. This is a standard HTTP 1.1 status code, and is returned, for example, instead of a 102 if the request has exceeded internal Network Automation timeouts.

Headers

The Network Automation API does not use any specialized HTTP headers.

Body

For the majority of API calls, the response body can be returned as either JavaScript Object Notation (JSON, content-type application/json), or Extensible Markup Language (XML, content-type application/xml). Some calls can optionally return JSON with special added attributes for use with the Network Automation ExtJS application (application/extjs). A few calls, such as those used to download files, will specify a particular response content-type; those are documented in the specific API call documentation.

In general, the choice of response format is up to the client. The details are in the table below, but the essential rules here are:

- GET requests return application/json by default, or you may specify the format by appending .json, .xml, or .extjs to the URI.
- POST requests return the content-type you posted in your request, unless that was a standard HTTP form encoding. In that case, the response format will be application/json, unless you specify the .xml or .extjs extension.

The table below enumerates the various combinations of request methods, extensions, and formats, and the resulting response format.

Request Method	Request URI Extension	Request Body Content Type	Response Body Content-Type
GET		n/a	application/json
GET	.json	n/a	application/json
GET	.xml	n/a	application/xml
GET	.extjs	n/a	application/extjs
POST		application/x-www-form-urlencoded	application/json
POST		multipart/form-data	application/json
POST		application/json	application/json
POST		application/xml	application/xml
POST		text/xml	application/xml
POST	.json	Any supported type	application/json
POST	.xml	Any supported type	application/xml
POST	.extjs	Any supported type	application/extjs

For application/json, the entire body of the response is an anonymous hash of name/value pairs, unless otherwise defined in the specific API call documentation. Responses using application/extjs will be the same, but will include some additional information in some cases.

For application/xml, the root element will always be the "response" element, with the children being elements tagged by the name and containing the values, unless otherwise defined in the specific API call's documentation.

In addition, with application/xml, special handling is required to support anonymous values for the basic JSON primitive types of Number, String, Boolean, Array, Object (hash) and null. To handle this, anonymous values are tagged with "item". Below are some example responses.

Description	JSON	XML

Response with a single return parameter named "foo" with value "bar".	{
	"foo": "bar"
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<foo>bar</foo>
		</response>

Response with a single return parameter named "foo" with integer value 123.	{
	"foo": 123
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<foo type="array">
		<item type="integer">123</item>
		<item>bar</item>
		<item type="bool">true</item>
		</foo>
		</response>

Response with a single return parameter named foo with a hash value containing a floating point number, a string, and a null.	{
	"foo": {
	"one": 12.3,
	"two": "bar",
	"three": null }
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<foo type="hash">
		<one type="float">12.3</one>
		<two>bar</two>
		<three nil="true"/>
		</foo>
		</response>

Response with two named parameters, the first "foo" with a string value "bar", the second "foofoo" with a null value.	{
	"foo": "bar",
	"foofoo": null
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<foo>bar</foo>
		<foofoo nil="true"/>
		</response>

Common compound data types, including hashes and arrays, have a specified XML format, as shown in the examples above. These are used for generic hashes and arrays, not for well-defined objects such as a "Device", which will be described in the specific API call.

Results Paging

Some API calls that return large result sets are limited to 1000 records per request to prevent unreasonable memory consumption on the Network Automation appliance. In this case, the response will include paging information, as in this example:

```
<response>
 <total type="int">1452</total>
 <start type="int">0</start>
```

```
 <limit type="int">1000</start>
 <current type="int">1000</start>
<devices type="array">
 <device>…</device> [ repeated 999 more times ]
</devices>
</response>
```

where:

`total` specifies the total number of records available for the data type.

`current` specifies the number of records returned in this group. No more than 1000 records are returned at one time.

`start` specifies the number of the first record in this group.

`limit` specifies the maximum number of records that was requested.

These API calls must be repeatedly queried, while modifying an input parameter to indicate which "page" to obtain. The Perl API provides this functionality without the script author needing to implement it specifically.

API calls that utilize results paging denote this in their documentation.

Standard Body by Status

In addition to the basic Content-Type, certain status codes have a standard payload, or body format. This allows a higher degree of code-reuse and simplified handling of the various data types within Network Automation.

102 Processing: The standard payload for this status indicates the progress of the processing, if available. If unavailable (i.e., how much processing has been completed is unknown), the "total" field will not be present, and the "done" field may or may not be present. Specific calls may add additional payload.

Description	JSON	XML

Format	{
	"total": <total to process>,
	"done": <done so far>
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<total>total to process</total>
		<done>done so far</done>
		</response>

Example (Known)	{
	"total": 50123,
	"done": 10456
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<total>50123</total>
		<done>10456</done>
		</response>

Example (Unknown)	{
	"done": 134
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<done>134</done>
		</response>

Example (Unknown)	{ }	<?xml version="1.0" encoding="UTF-8"?> <response/>

200 OK: The standard payload is nothing for this status code. Many API calls will return a payload specific to the call.

201 Created: The standard payload for this status is the model and ID of the newly created entity, along with a URI to retrieve the details of the entity, and the actual newly created entity. In addition, the URI will be in the Location: HTTP header. Specific calls may add additional payload.

	JSON	XML

Format	{
	"model": "<model name>",
	"id": <id value>,
	"uri":
	http://<ip>/<plural-model-name>/<id>,
	"<model name>": { … }
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<model>model name</model>
		<id>id value</id>
		<uri>
		
		http://<ip>/<plural-model-name>/<id>.xml
		</uri>
		<model name> … </model name>
		</response>

Example	{
	"model": "script",
	"id": 45,
	"uri":
	"http://10.1.1.1/scripts/45"
	"script": { "id":45, … }
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<model>script</model>
		<id>45</id>
		<uri>
		http://10.1.1.1/scripts/45.xml
		</uri>
		<script>
		<id type="int">45</id>
		...
		</script>
		</response>

202 Accepted: The standard payload for this status is the same as for the 201 Created, except without the actual model data; it is the model and ID of the newly created entity, along with a URI to retrieve the details of the entity. The difference is that for a 201 Created code, the entity has already been created. For a 202 Accepted, the entity is in the process of being created. Thus, a request to the provided URL may return a 102 Processing instead of a 200 OK.

Just as with the 201 Created, the URI will be in the **Location: HTTP** header.

Specific calls may add additional payload.

	JSON	XML

Format	{
	"model": "<model name>",
	"id": <id value>,
	"uri":
	"http://<ip>/<plural-model-name>/<id>"
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<model>model name</model>
		<id>id value</id>
		<uri>
		
		http://<ip>/<plural-model-name>/<id>.xml
		</uri>
		</response>

Example	{
	"model": "script",
	"id": 45,
	"uri":
	"http://10.1.1.1/scripts/45"
	}
		<?xml version="1.0" encoding="UTF-8"?>
		<response>
		<model>script</model>
		<id>45</id>
		<uri>
		http://10.1.1.1/scripts/45.xml
		</uri>
		</response>

301 Moved Permanently: No payload is present for this status code. The Location: header will indicate the redirect location. Typically used for a deprecated method that has a direct replacement.

400 Bad Request: The standard payload for this status code is an error code and message. The message may change based upon localization, therefore automated clients should use the error code. This basic format is called the error response format and is used for other 4xx status messages as well. See the following section, *Standard Error Codes* for details on the widely-used error responses.

	JSON	XML
Format	{ "error": "<error code>", "message": "<error message>" }	<?xml version="1.0" encoding="UTF-8"?> <response> <error>error code</error> <message>error message</message> </response>
Example	{ "error": "general/resource-in-use" , message: "The requested shared resource is currently in use. Please try your request again in a few mintues." }	<?xml version="1.0" encoding="UTF-8"?> <response> <error>general/resource-in-use</error> <message>The requested shared resource is currently in use. Please try your request again in a few mintues.</message> </response>

The specific error codes returned will depend upon the API call, but certain error codes have pre-defined formats, as described below.

403 Forbidden: The standard payload is the error response format, with one of the following two error codes: “security/access-denied” or “security/authentication-required”.

404 Not Found: No standard payload is present for this status code, although some specific calls may define one.

409 Conflict: The standard payload is the error response format, which will include information on how to resolve the conflict.

410 Gone: This status code indicates that the API call requested is no longer available. A standard error/message payload may be present explaining the reason and providing direction for implementing the same functionality with the newer API.

413 Request Entity Too Large: The standard payload is the error response format, which will include details of what aspect of the request was too large (size or length of time, for example).

Standard Error Codes

The following error codes are used in the error response format throughout the application, whenever a 4xx status is returned. Individual API calls may also add their own payloads to these responses.

JSON	XML
general/cannot-process-request	A generic error response; the message should detail the specific meaning. This is used when no reasonable special handling by the client is possible.
general/validation-failed	The inputs provided in the request do not meet the criteria for this API call.
general/resource-in-use	The requested shared resource is not available at this time; the request should be tried again in a few minutes.

JSON	XML
security/access-denied	The requestor is currently authenticated, but does not have sufficient privileges to access the specified resource.
security/authentication-required	The requestor is not currently authenticated and must authenticate and retry the request.

Additional Payloads by Error Code

general/validation-failed: Currently, this is the only error type that will have any difference in JSON and ExtJS versions.

	JSON	ExtJS	XML

Format	{
	"error": "general/validation-failed",
	"message": "<error message>",
	"fields": { <hash of field name, array of failure messages> }
	}
		{
		"success": false,
		"messages": [
		<array of strings>
		]
		}
			<?xml version="1.0" encoding="UTF-8"?>
			<response>
			<error>error code</error>
			<message>error message</message>
			</response>

Example	{
	"error": "general/validation-failed",
	"message": "The policy could not be created.",
	"fields": { short_name: ['is already taken', 'must be 12 or fewer characters'], name: 'is too long'}
	
	}
		{
		"success": false,
		"messages": ['Short name is already taken','Name is too long']
		}
			<?xml version="1.0" encoding="UTF-8"?>
			<response>
			
			<error>general/validation-failed</error>
			<message>The policy could not be created.</message>
			<fields type="hash">
			<short_name type="array">
			<item>is already taken</item>
			<item>must be 12 or fewer characters</item>
			</short_name>
			<name type="array">
			<item>is too long</item>
			</name>
			</fields>
			</response>

`General/resource-in-use`: The additional payload is optional for this error type; it will be included when available.

	JSON	XML
Format	`{ "error": "general/resource-in-use", "message": "<error message>", "model": "<model name>", "id": <id value>, "retry": <seconds> }`	`<?xml version="1.0" encoding="UTF-8"?> <response> <error>general/resource-in-use</error> <message>error message</message> <model>model name</model> <id>id value</id> <retry>seconds</retry> </response>`
Example	`{ "error": "general/resource-in-use", "message": "A job is already running against this device." "model": "device", "id": 53534634, "retry": 60 }`	`<?xml version="1.0" encoding="UTF-8"?> <response> <error>general/resource-in-use</error> <message>A job is already running against this device.</message> <model>device</model> <id>53534634</id> <retry>60</retry> </response>`

Extended JSON Specifications

The Network Automation API uses standard JSON as described in RFC 4627, with the following additional convention. For Object primitives, the key "_class" (an underscore followed by the word "class") is reserved to indicate the class name of the object. This is equivalent to the model-type rule described in the XML specification below.

Extended XML Specifications

As noted above, JSON provides a set of primitive types. XML does not provide such a mechanism. Therefore, below is the grammar used to represent these types in XML. To properly read this grammar, note that in instance:

- `NCName` refers to a "non-colonized" name; that is, and XML element name without any namespace prefixing (see http://www.w3.org/XML/Schema.html for more). When used as the opening and closing of an element, it of course must follow XML standards and use the same name for the opening and closing tags.
- Use `=` to denote the definition of a grammatical rule
- Use `|` to mean "or"
- Use `[ ]` indicates that the contents is optional
- Italics indicates a rule name, not literal text
- `*` indicates that the literal or rule-name that follows may be repeated 0 or more times
- Use `( )` for association
- `/ /` indicates a Perl-style regular expression
- By default, the root element is treated as a hash, and other elements as strings.

- Example: `foobar = foo *(, ( foo | bar) )` defines a rule, “foobar”, that validates a string that starts with “foo” and then is followed by “foo” and/or “bar” zero or more times, separated by commas (such as: `“foo,bar,foo,foo,bar,foo,bar”`)

Extended XML Grammar

```
object = <object-type> *name-value </object-type> | <NCName [ type="object-type" ]>
*name-value </NCName>
name-value = <NCName [ type="type" ] [ nil="boolean" ]> value </NCName>
type = string | integer | int | float | boolean | bool | hash | array | object-type | nil
object-type = hash | model-type
model-type can be any of the models documented in Tools > Network > API Documentation.
value = string | integer | float | boolean | object | array | nil
string is a standard XML text node.
integer = /^[-+]?[0-9]+$/
float = /^[-+]?[0-9]+(\.[0-9]+)?$/
boolean = true | false
true = /^true$/i | /^yes$/i | y | Y | 1
false = /^false$/i | /^no$/i | n | N | 0
array = <array> *arrayvalue </array> | <NCName type="array"> *value </NCName>
arrayvalue = name-value(NCName="item")
nil = <nil/> | <NCName nil="true"/>
```

Perl API

For API call details, the descriptive table of Perl API calls can be located at the following:
Click the **Tools** icon and choose **Network –› API Documentation** and click the API List link.

Note: When using API calls, replace `BASE` in any table entry with the `base_uri` value returned by the `/api/base_uri` call.

The Perl API functions as a wrapper around the API Core. This wrapper binds the XML output to Perl objects.

Tip: Infoblox recommends familiarity with the API Core before working with the Perl API.

Installing the Perl API

The following are required to use the Perl API:

- Unix, Linux or Windows operating system
- Perl 5.8.8 or later for Unix/Linux, Mac OS X or ActiveState's Active Perl for Windows

Perl modules:

- libwww-perl
- JSON
- Scalar::Util =› v. 1.18

These can be installed on Debian / Ubuntu using:

```
% sudo apt-get update
% sudo apt-get install libwww-perl libjson-perl
```

These can be installed on RedHat / CentOS / Fedora using:

```
% sudo yum install perl-libwww-perl perl-JSON
```

Optional Perl modules include:

- JSON::XS
- YAML
- XML::Simple

Install optional modules as follows:

```
% sudo apt-get install libjson-xs-perl libxml-simple-perl libyaml-perl
```

These packages should be available on all recent versions of either Debian or Ubuntu. They can also be installed on RedHet/CantOS/Fedora using the following:

```
% sudo yum install perl-XML-Simple
```

Some of the noted packages may not be available on all recent RPM-based Linux distributions; if that is the case for yours, then the CPAN method (see next) can be used. Use the CPAN shell:

```
% cpan LWP JSON NetAddr::IP Scalar::Util
```

For optional modules:

```
% cpan JSON::XS XML::Simple YAML
```

Once the prerequisites are met, extract the Network Automation API distribution, and install:

```
% tar zxf NetMRI-API-3.0.0.0.tar.gz
% cd NetMRI-API-3.0.0.0
% perl Makefile.PL
% make
% make test
% sudo make install
```

On Windows, **nmake** or **dmake** can be used instead of **make**. Nmake is provided with Microsoft Visual Studio.

Authentication

When your script connects to Network Automation, it must pass the URL, user name and password to the client object. The URL and credentials can be passed into the constructor when connecting to Network Automation:

```
my $client = new NetMRI::API({
  api_version => 3.0,
  url => 'http://netmri',
  username => 'admin',
  password => 'secret',
});
```

Or you can create a .netmri.yml (requires installation the YAML module is installed) or .netmri.json in your home directory.

For .netmri.yml:

```
---
url: http://netmri
username: admin
password: secret
```

For .netmri.json

```
{"password":"secret","url":"http://netmri","username":"admin"}
```

The .netmri.yml or .netmri.json file should be appropriately protected if you include a password.

Alternatively, use cookie-based authentication with a .netmri.yml config similar to the following:

```
---
url: http://netmri
username: admin
persistent: 1
```

Run `netmri-api-login` to interactively login before running any other scripts that use the netmri. Netmri-api-logout can be used to logout.

```
grunion% netmri-api-login
http://admin@172.23.23.70 password:
```

```
grunion% netmri-api-logout
grunion%
```

Using the Perl Objects

The Perl API is written using object-oriented Perl. All interactions with the Network Automation device are through a client class called NetMRI::API. An instance of this class must be created before interactions can be performed. To open a connection with Network Automation:

```
my $client = new NetMRI::API({ api_version => 3.0 });
```

The api_version must be included, and ensures that your script will use the same interface even as the Core API evolves over time.

Because no other parameters are passed when creating the object, the client will attempt to connect and authenticate with Network Automation using the configuration file from the previous section.

For each data type-such as “device,” “interface” or “VLAN”-there is a broker class. This class contains all the routines that query, summarize or otherwise act on sets of data objects. The NetMRI::API::Broker::ConfigRevision, for instance, is a broker that can be used to query for specific list of configuration files. Files can be searched across all active saved and running configuration files.

Example 1

Suppose you want to list all routing devices on your network. Network Automation provides (at least) two different ways to identify these devices. First, you can search by device type; querying for devices with types “Router” and “Switch-Router” will list all these devices. To do this, first create a device broker using the `NetMRI::API::Client`:

```
my $broker = $client->broker->device;
```

Next, query for devices having Device Type of “Router” or “Switch-Router”:

```
my @devices = $broker->index({ DeviceType => [“Router”, “Switch-Router”]});
```

This will make one or more HTTP requests to retrieve all of the devices from the Network Automation which match the query. When all records have been retrieved, the @devices array contains the complete list of “Device” objects. You can access the fields of the Device object using accessor methods. This loop, for example, prints a list of all the devices returned, including each devices's name, IP address and device type:

```
foreach my $device (@devices)
{
  print $device->DeviceName, ' ', $device->DeviceIPDotted, ' ', $device->DeviceType,
"\n";
}
```

The other accessor methods are documented in the Perl documentation for the `NetMRI::API::Remote::Device` class:

```
% perldoc NetMRI::API::Remote::Device
```

As noted, there is an alternative way to identify routing devices. The Device model provides a flag, **RoutingInd**–which is set to 1 for any device from which Network Automation has successfully collected a routing table. The query would look like the following:

```
my @devices = $broker->index({ RoutingInd => 1 });
```

Additional documentation

Use `perldoc` to read documentation for individual objects. For example:

```
% perldoc NetMRI::API
```

`perldoc` provides a complete list of methods available in each Perl object.

Another useful document is NetMRI::API::Index, which lists all of the available broker and remote object classes.

```
% perldoc NetMRI::API::Index.
```

HTML versions of this documentation are also available in the **Network Automation Tools –› Network** section **–› API Documentation –› Perl API**.

Example scripts

Several example scripts are included with the NetMRI::API distribution in the /examples directory.

Migration to API Core Version 3.0

Migrating scripts from older 2.x API Core versions should be relatively painless. For scripts that are unlikely to see new development and just need to work, consider using the `NetMRI::API::Client` compatibility layer. To do this you don't need to change anything in your script. You can even take advantage of some features of the new interface while using the compatibility layer.

There are some advantages to migrating old code to use the new interface. The new interface is somewhat more efficient without the compatibility layer, and is more succinct for common tasks, such as operating over a list of objects.

Api_version is Required

The constructor parameter `api_version` to `NetMRI::API` is required. By providing an `api_version` you guarantee your script will continue to function as expected as the Network Automation API evolves in the future. You may pass 'auto' as the argument to indicate that you really do want the API to detect the latest version available to both the Perl API and the Network Automation system. This indicates to the API that you understand that your script may behave differently when connecting to future versions of Network Automation.

```
my $client = new NetMRI::API::Client;
```

becomes either

```
my $client = new NetMRI::API(api_version => 'auto' );
```

or

```
my $client = new NetMRI::API(api_version => 2 );
```

Naming Conventions

The 3.0 API uses the most common Perl naming conventions. Constructor arguments and accessor methods are lower-case under-score separated.

```
my $client = new NetMRI::API::Client(UserName => 'admin', Password => 'secret');
```

becomes

```
my $client = new NetMRI::API({ username => 'admin', password => 'secret',
api_version => 3.0 });
```

Passing Arguments as a Hash Reference

The new 3.0 API encourages you to pass arguments as a hash reference instead of a list (which later gets converted into a hash anyway). Passing arguments as a list is still supported to maintain backward compatibility, but is deprecated.

```
$client->index( DeviceID => 2 )
```

becomes

```
$client->index({ DeviceID => 2 })
```

Accessors

Remote object classes have accessor functions for retrieving field values, instead of treating them as a hash. The old behavior of treating these objects as a hash is supported because the way remote object classes are implemented, but depending on this behavior is deprecated.

```
$device->{DeviceID}
```

becomes

```
$device->DeviceID
```

Broker Objects

Broker objects are cached by the $client object. This simplifies code when the same broker objects are used in different parts of the code.

```
my $broker = $client->get_broker('Device');
my $device1 = $broker->show( DeviceID => 1 );
my $device2 = $broker->show( DeviceID => 2 );
```

becomes

```
my $device1 = $client->get_broker('Device')->show( DeviceID => 1 );
my $device2 = $client->get_broker( 'Device')->show( DeviceID => 2 );
```

In addition, a new syntax for retrieving broker objects can be used:

```
my $device1 = $client->broker->device->show( DeviceID => 1 );
my $device2 = $client->broker->device->show( DeviceID => 2 );
```

List Methods

Some methods, such as index, return a list of remote objects. In the old interface you could pass `RetrieveAllPages => 1` as an argument to these list methods to get all objects, if the number of objects is greater than the page size. In the new interface you can get the list of all objects by calling the method in list context.

```
my $response = $client->get_broker('Interface')->index(__RetrieveAllPages => 1);
    foreach my $interface (@{ $response->{interfaces} })
    {
    print $interface->ifName,"\n";
    }
```

becomes

```
    foreach my $interface ($client->broker->interface->index)
    {
    print $interface->ifName, "\n";
    }
```

Attaching Files

The 3.0 API retains the following syntax from previous releases for attaching files.

```
use NetMRI::API::Client;
    my $client = new NetMRI::API::Client;
    $client->get_broker('DiscoverySetting')->import( file=> 'LOCAL_FILE:file.txt',
    import_type => 'range',
);
```

becomes

```
use NetMRI::API qw( netmri_api_file );
    my $client = new NetMRI::API::Client;
    $client->broker->discovery_setting->import( file=> netmri_api_file 'file.txt',
    import_type => 'range',
);
```

If `netmri_api_file` is too verbose you can also simply import a file instead.

```
use NetMRI::API qw( file );
    my $client = new NetMRI::API::Client;
    $client->broker->discovery_setting->import( file=> file 'file.txt', import_type =>
    'range',
);
```

In the 3.0 API, use `NetMRI::API::FindIt`for interacting with the Network Automation appliance's FindIt interface.

```
use NetMRI::API::Client;
    my $client = new NetMRI::API::Client;
    my $findit = $client->get_broker('FindIt');
    my @result = $findit->find('foo');
```

becomes either

```
use NetMRI::API;
    my $client = new NetMRI::API(api_version => 3.0 );
    my @result = $client->findit('foo');
```

or:

```
use NetMRI::API::FindIt;
    my $findit = new NetMRI::API::FindIt( url=> 'http://netmri', username => 'admin',
    password => 'secret',
    );
my @result = $findit->find('foo');
```