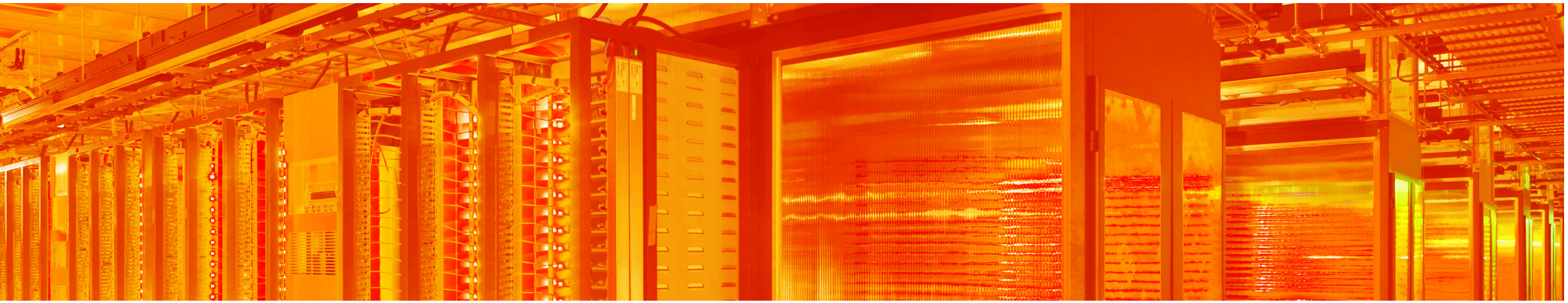
REST API Guide



NX500 Controller
VX1048 Switch
VX3048 Switch

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

vello SYSTEMS

Notice

Vello believes the information in this publication is accurate as of its publication date. However, the information is subject to change without notice. **THE INFORMATION IN THIS PUBLICATION IS PROVIDED "AS IS." VELLO MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

Use, copying, and distribution of any Vello software described in this publication requires an applicable software license.

For the most up-to-date regulatory document for your product line, please refer to your specific agreements or contact Vello Technical Support at support@vellosystems.com.

The information in this document is subject to change. This manual is believed to be complete and accurate at the time of publication and no responsibility is assumed for any errors that may appear. In no event shall Vello Systems be liable for incidental or consequential damages in connection with or arising from the use of the manual and its accompanying related materials.

Copyrights and Trademarks

Vello Systems, VellOS, Vello Systems NX500, Vello Systems VX1048, and Vello Systems VX3048, are trademarks of Vello Systems, Inc. Please review the Vello Corporation Trademarks at http://www.vellosystems.com for additional/updated product name and trademark information. All other trademarks are the property of their respective owners.

Contact Information

Vello Systems, Inc.
1530 O'Brien Drive
Menlo Park, CA 94025

Phone: 650-324-7600
Fax: 650-324-7601
Toll-free: 1-866-MY-GIGES (1-866-694-4437)

Email: support@vellosystems.com

Website: www.vellosystems.com

Table of Contents

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

This page intentionally left blank.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

1: Getting Started

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

API Overview

An application programming interface (API) is a protocol that allows software components to communicate with each other. A protocol often includes a library of specifications for routines, data structures, object classes, and variables. Libraries can take many forms, including online resources, sample code, or even hardcopy manuals.

APIs that enable web communication may be either server-side or client-side.

- Server-side APIs reside within web servers to provide programmatic interfaces to defined request-response message systems (clients). These interfaces are typically expressed in JSON or Extensible Markup Language (XML), and are exposed via the web (most commonly via HTTP-based web servers).
- Client-side APIs reside within web browsers.

Note: Web applications that use multiple web API protocols are called mashups.

About RESTful APIs

REpresentational State Transfer (REST) is a software architectural design style for creating interfaces between distributed systems that is widely used in web API design. Styles and protocols include defined constraints (requirements); however, styles give users the freedom to meet those requirements in any way they choose. End users experience REST APIs as they browse the web as follows:

1. A user's browser provides access to a network of web pages (virtual state-machines). One can think of each page as a state within an application.
2. Clicking a link (state transition) within an application causes the next page (next application state) to transfer to the user and render for use.
3. A global identifier (such as a Uniform Resource Identifier, or URI) references each resource of specific information that resides on a server.
4. Network components communicate via a standardized interface (such as HTTP), and exchange representations of these resources, in order to interact with and manipulate these resources.

REST is essentially a method for bringing together and applying many existing, defined, and recognized software standards to create efficient, scalable, and secure software solutions to integrate information exchange across distributed systems. A *RESTful* software implementation is one that follows the REST method. A RESTful web API (also called a RESTful web service) is a web API that is coded according to the REST design style and implemented using HTTP methods.

A web application with a RESTful web API needs to know only the following to interact with a resource:

- Required URI
- Required action (HTTP method)

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

- Format (representation) of returned information (such as an HTML, XML, or JSON document, an image, plain text, or any other content)

The application need not know whether caches, proxies, gateways, firewalls, tunnels, or anything else exist between it and the server that actually holds the information.

REST defines six complementary design constraints:

- **Separation:** Servers house data and/or shared applications, but do not store methods to display data. Clients contain the display methods and applications needed to view and interact with the data provided by the servers. Separating data storage from data display facilitates scalability and extensibility on both sides. Clients and servers communicate via APIs.
- **Statelessness:** A server need neither know nor store ("remember") the client state. Each *call* (state transition) from a client carries all of the information needed for the server to provide the client with data that match the requested new state. For example, a web server need not track what individual web page each user is currently on; it need only provide the next web page requested by each user as they trigger calls by clicking hyperlinks. The server is thus stateless and free of a potentially huge computing burden.
- **Caches:** Clients may *cache* (store) designated data received from servers, thus improving network and server performance by reducing the number of data calls to the server. The risk of caching is that data may become obsolete while the client is still using it. A well-designed, well-managed data schema can mitigate this risk.
- **Layers:** Clients need not know the network structure lying between them and the server. Different servers may provided different portions of the requested data, and other devices may provide additional functions, such as load-balancing, routing, or security.
- **Executables (optional):** A server may supply an application and/or executable code along with the data being returning the client. This additional functionality is usually temporary, with the executable being removed when the user session reaches some closing point.
- **Uniformity:** A uniform interface allows network data to transfer using a standard format. REST uses four fundamental principles for creating uniform client/server interfaces:
  - **Resource identification:** Each data source must have a unique identifier, such as a URI. A resource that receives a call for data returns a representation of the requested data. For example, assume that a user clicks a link to look at a table that contains listings drawn from several databases at once. In this example, each named piece of data has a URI. The server(s) storing that URI send only the HTML representations of the exact data needed to fulfill the request; they do not send the entire database, nor do they render the table the user is viewing.
  - **Resource manipulation:** A client receives data-handling permissions (such as search, read, edit, and delete) from the server along with the data. This allows a user with the appropriate permissions to interact with and change the data being stored on the source server(s). REST requires permissions to accompany the data, but does not define how those permissions must be handled.
  - **Self-descriptive messages:** Data for each state change includes metadata that instructs the client how to display that data.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

- **Hyperlink-driven client state changes:** The client is aware of the hyperlinks currently available to the user on their current display, but is unaware of the actions that occur if the user clicks one of those links. The client only becomes aware of the results of a state change upon receiving return data from the server.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

About the Vello REST API

The OpenFlow-enabled NX500 Controller from Vello Systems, Inc. is deployed with a single image that consists of the Linux operating system with several custom-built user-space applications that form a platform for programmatic network control. This platform has three externally-facing interfaces that each offer a particular set of interactions and operations with some overlap:

- **Command Line Interface (CLI):** Custom console that uses a restricted command set. This interface allows initial device bring-up and configuration, root access, and recovery.
- **Unified Bandwidth Manager (UBM):** HTTPS graphical user interface. This interface provides an efficient, intuitive way for network administrators to perform routine tasks, such as defining packet flow policies and monitoring packet traffic. UBM leverages the REST API.

Note: Refer to the User Guide for complete CLI and UBM instructions.

- **RESTful Web API (REST API):** Accessed via HTTPS. This interface allows the creation of custom networking solutions beyond CLI and UBM capabilities, such as setting the NX500 Controller IP address. You may also perform routine CLI- and UBM-accessible tasks, such as defining flow policies.

Vello REST API System Requirements

The Vello REST API comes pre-enabled on the NX500 Controller. The minimum system requirements for running the API are the same as the requirements for running the UBM.

The computer used to access UBM must meet the following minimum requirements in order function as an effective NX500 Controller client:

- **Operating System:** Windows 7 or 8
- **Browser:** Firefox 18.0 to 20.0; Chrome 24.0 to 26.0; or Internet Explorer 9 or 10
- **Minimum resolution:** 1024x768

In addition, the NX500 Controller must be:

- Powered on
- Network-accessible by the computer being used to access the REST API
- Configured with at least one Vello VX1048/VX3048 switch that is in **Active** status.

See the *User Guide* for bring-up, connection, and configuration instructions.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Accessing the Vello REST API

To access the Vello REST API:

Verify that the system being used for access meets all of the system requirements described in the previous section.

1. Open a web browser and navigate to `https://<A.B.C.D>/`, where `<A.B.C.D>` is the Management IP address of the NX500 Controller.
2. The NX500 Controller redirects the browser to the Web services page.
3. Enter your user name in the **Login** field (case sensitive). The default user name is `admin`.
4. Enter your password in the **Password** field (case sensitive). The default password is `vello123`.

Note: Remember that user authentication is identical for the UMB and API; a user with access to one has access to the other.

5. After logging in, navigate to `http://<A.B.C.D>/rest/v1.0/switch`, where `<A.B.C.D>` is the Management IP address of the NX500 Controller.

   The system will respond with a message similar to the following:


```
{"items":[{"id":"SW0000B0D2F5052087","inventory":{"hostname":"SW0000B0D2F5052087","data-plane-id":"0x0000b0d2f5052087","description-mfr":"Vello Systems, Inc, 1530 Obrien Dr, Menlo Park, CA 94025.","description-hw":"VX3048","description-sw":"Jun  5 2013 11:15:00","description-dataplane":"","serial-num":"1234567890123456"},"ip-config":{"mode":"static","ip-addr":"172.18.254.155","sub-net":"0.0.0.0","gateway":"0.0.0.0"}
```


AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

About This Manual

This section describes the formatting conventions and information contained in this manual.

Formatting Conventions

This manual uses several formatting conventions to present information of special importance.

Lists of items, points to consider, or procedures that do not need to be performed in a specific order appear in bullet format:

- Item 1
- Item 2

Procedures that must be followed in a specific order appear in numbered steps:

1. Perform this step first.
2. Perform this step second.

Specific keyboard keys are depicted in square brackets and are capitalized, for example: [ESC]. If more than one key should be pressed simultaneously, the notation will appear as [KEY1]+[KEY 2], for example [ALT]+[F4].

Interface elements such as document titles, fields, windows, tabs, buttons, commands, options, and icons appear in **bold** text.

Menus and submenus have the notation **Menu>Submenu**. For example, "Select **File>Save**" means that you should first open the **File** menu, and then select the **Save** option.

Specific commands appear in standard `Courier` font. Sequences of commands appear in the order in which you should execute them and include horizontal or vertical spaces between commands. The following additional formatting also applies when discussing Command Line Interface (CLI), cURL, and Python commands:

- Actual commands appear in plain `Courier` font. Type these commands as shown.
- CLI responses from the system appear in bold **`Courier`** font.
- Optional values appear in square brackets, such as `[value]`. Do not include the brackets when adding an optional value to a command. If there is more than one optional value, you will see a vertical pipe between individual choices (such as `[yes|no]`). You may select either none or one of the optional values.
- Variable values appear inside carets, such as `<severity>`. In this case, replace the variable with a specific value from the list of available options for that variable. Do not include the carets when entering the value.
- Mandatory inputs where you must select one of two or more specific values appear in carets with a vertical pipe between individual options (such as `<tcp|ssl>`). In these cases, you must select one of the values when entering the command.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

- Number ranges appear inside carets, such as <1-10>. In this example, you may input any number from 1-10. Do not include the carets when entering the value.

This manual also contains important safety information and instructions in specially formatted callouts with accompanying graphic symbols. These callouts and their symbols appear as follows throughout the manual:

CAUTION: CAUTIONS ALERT YOU TO THE POSSIBILITY OF EQUIPMENT DAMAGE AND/OR PERSONAL INJURY IF THESE INSTRUCTIONS ARE NOT FOLLOWED.

Note: Notes provide helpful information.

Organization

This manual contains the following chapters:

- **1 - Getting Started:** Introduces APIs at a high level and outlines the conventions and formatting used in this manual. It also includes contact information for Vello Systems, Inc.
- **2 - REST API Structure:** Provides a high-level overview of the REST API (see <xref>).
- **3 - NX500 Configuration:** Describes using the REST API to configure the NX500 Controller (see <xref>).
- **4 - JSON:** Lists the JSON elements for working with nodes, flows, and the network topology (see <xref>).
- **5 - cURL:** Contains sample cURL scripts. You may use cURL scripts to make REST API calls by providing basic parameters derived from the httplib and json libraries (see <xref>)
- **6 - Python:** Contains sample Python scripts. You may use Python scripts to make REST API calls by providing basic parameters derived from the httplib and json libraries (see <xref>).

Note: Detailed httplib and JSON instructions are beyond the scope of this manual, as are cURL and Python programming instructions.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Additional Information

This section lists related documentation and provides information on contacting Vello Systems, Inc.

Related Documentation

Please refer to the following documents for additional information:

- **Hardware Specification:** This manual contains detailed specifications for the Vello NX500 Controller and the VX1048 and VX3048 switches. It also lists the Field Replaceable Units (FRUs) for each device and provides ordering information.
- **User Guide:** This manual describes installing, bringing up, and configuring the Vello NX500 Controller and VX1048/VX3048 switches. It also covers the CLI, UBM, and troubleshooting.

Contact Information

You may contact Vello Systems, Inc. at the following addresses/phone numbers:

Corporate Headquarters
1530 O'Brien DR
Menlo Park, CA 94025
T. +1 650-324-7600
USA

United Kingdom
66 Chiltern STT
London W1U 4JT
T. +44 (0) 7885725192

Germany
Forstring 96
63225 Langen
T. +49 6103 5095225

Please see the *User Guide* for information about how to obtain support and warranty service.

This page intentionally left blank.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

2: Vello REST API Structure

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Definitions

This section defines some basic terms that are used throughout this manual.

Software Defined Network

A *software-defined network* (SDN) uses software to decouple network control from hardware in favor of a software application called a controller. This provides much greater flexibility and control at a much lower cost than traditional hardware-based methods.

Node

A *node* is a port on a VX1048 or VX3048 switch. It may also specify either a single IP address (such as a PC) or a range of IP addresses that represent a subnet that is allowed into and/or out of that port. Thus, a node can be defined by either:

- A combination of IP address/netmask of a host/network and a physical VX port through which the host/network can be reached.
- A VX switch port only. This represents any network entities that can be reached through the VX port regardless of Layer 2 or Layer 3 addressing.

Flow

A *flow* is a logical network route through the SDN, between nodes. The act of defining a flow means that traffic is both permitted and expected to flow between the nodes. The NX500 Controller automatically computes the individual physical ports and switch links through the SDN that are required to create the flow, and programs the VX1048 and/or VX3048 switch(es) appropriately. A flow may have the following properties:

- **Direction:** A flow may be *unidirectional* (one way) or *bidirectional* (both ways) between a pair of nodes.
- **Bandwidth:** All flows require reserved (guaranteed) bandwidth. If the user does not supply a value, the bandwidth reservation will default to the node port speed. This bandwidth is reserved throughout all of the physical links that comprise the flow. The flow cannot be created if the specified bandwidth is unavailable.

  When creating a flow, the NX500 Controller updates the network topology with new bandwidth reservations to ensure that future flows will not impinge on the bandwidth reserved for that flow. Removing a flow frees up the reserved bandwidth for use by other flows.
- **Unicast or Multicast:** See the following sections.

Unicast Flow

A *unicast* flow is a connection between two specific nodes. Data packets received on a single source node flow to a single destination node. You may further filter allowable traffic for a flow by defining the IP address and netmask of the ingress and egress nodes attached to the

VX1048/VX3048 ports. Unicast flows may be either unidirectional or bidirectional.

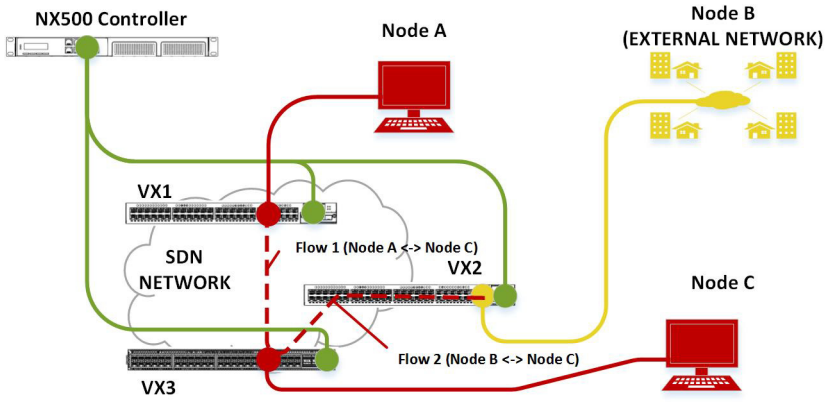


Figure 2.1: Unicast flows example

Multicast Flow

A *multicast* flow exists when a single source node forwards data to multiple destination nodes. The source node for a multicast flow does not use a multicast IP address; however, if a flow's destination node uses a multicast IP address, then the flow may have additional destination nodes (listener nodes) attached to it. Each additional listener node shares a common multicast IP address, called the *multicast group IP address*. Multicast flows must be unidirectional.

To create a multicast flow:

1. Define a node as the multicast source, including the switch, port, and unicast IP address.
2. Define a node as the multicast listener, including the switch, port, and the IP address of the multicast group.
3. Create a unidirectional flow from the multicast source to the multicast listener.
4. Add additional destinations to the flow that you just created.

   Creating this multicast flow returns a flow ID. You may now add more multicast listeners to the same flow by creating a new multicast listener node. Each multicast flow uses a single bandwidth when delivering information to listeners; you may not specify bandwidth on a per-listener basis.
5. Use the REST API **flow-target** URI (see <xref>) to add the listener node to the multicast flow using the flow ID and a node ID.

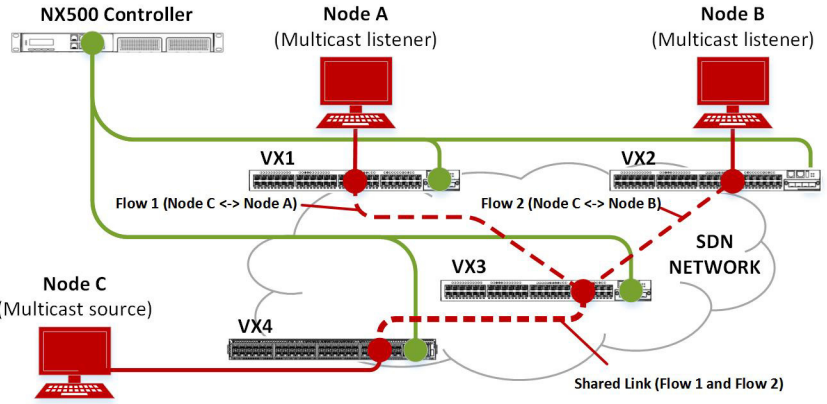


Figure 2.2: Multicast flow example

Note: You can use the same flow ID and node ID to remove a multicast listener.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Path

A *path* is the actual physical route that flows traverse through switches. a path is defined by two aspects:

- Switch ID and data ingress port ID (the physical port through which the data entered the switch)
- Switch ID and data egress port ID (the physical port through which the data exited the switch).

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Components and Capabilities

The Vello REST API interacts with the Vello OpenFlow SDN in terms of nodes and flows to:

- configure node policies,
- configure flow policies, and
- inspect the network *topology* (physical state).

You may also use the REST API to set the NX500 Controller Management IP address and whether that IP address is static or operates under Dynamic Host Configuration Protocol (DHCP). A static management IP addresses requires the following:

- Management IP address
- Netmask
- Default gateway

The NX500 Controller will use defaults for these values if you set the management IP address to DHCP (see <xref>).

You may set the NX500 Controlplane IP address (address of the ports used to manage VX1048 and/or VX3048 switches) using either the REST API or the CLI.

Flow Computation

The Vello flow computation algorithm computes flows based on the following priorities:

1. Select flows that satisfy the specified bandwidth requirements.
2. Select flows with the lowest number of hops that satisfy the specified bandwidth requirements.
3. Avoid duplication of packets, provided that the preceding two criteria are satisfied.

Note: The number of hops equals the number of switches the data traverses on the way from the source node to the destination node.

Topology Discovery

Topology discovery refers to the ability of the NX500 Controller to discover all of the switches in the network and the connections between neighboring switches for the purpose of flow computation. The following events trigger topology discovery:

- **New VX swit**ch: The NX500 Controller automatically discovers all of the neighbors and the network links between the neighbors when a new VX1048/VX3048 switch connects.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

- **VX switch port transitions to the "UP" state:** When a switch port enters the "UP" state, the NX500 Controller determines whether the port is connected to a neighboring switch and discovers the neighboring switch port.
- **VX port transitions to the "DOWN" state:** When a switch port moves to the "DOWN" state, the NX500 Controller automatically removes any switch link to a neighboring VX1048/VX3048.
- **VX disconnects from NX500 Controller:** The NX500 Controller determines that a switch is disconnected from the management network if:
  - the (TCP/TLS) connection to the controller from the switch is reset or closed, or
  - the switch fails to respond to an OpenFlow echo request generated by the NX500 Controller.

  The NX500 Controller initiates an OpenFlow echo request after 60 seconds without a message from the VX switch. The NX500 Controller considers that management connection disconnected if no reply comes within 60 seconds and removes the affected switch from the network topology. See the *User Guide* for troubleshooting information.

Communications

The following diagram illustrates how REST API components communicate among each other.

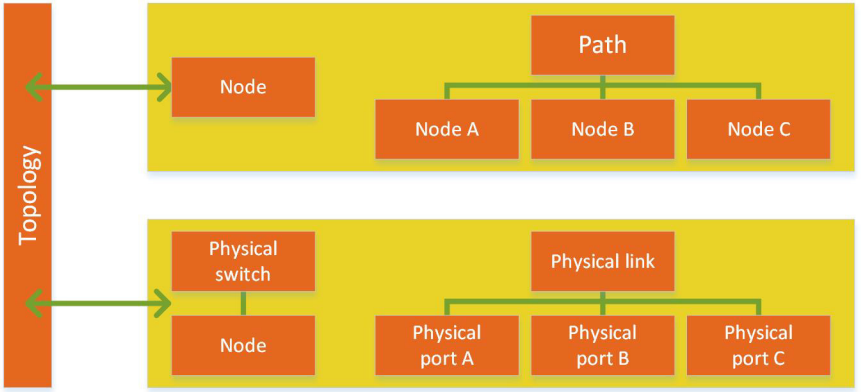


Figure 2.3: Communications between REST API components

Statistics and Reporting

Node and flow statistics are available for the logical network. Port statistics describe the physical network. Obtaining these statistics can be done via pull or push, as follows:

- **Pull:** You may query the NX500 Controller for statistics using the REST API.
- **Push:** You can configure the NX500 Controller to automatically send statistics to the management network using SFLOW.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Type	Fields	Default Behavior
Flow Stats Statistics related to traffic on a logical flow	• Total number of packets • Total number of bytes • Packets per second averaged over the configured interval • Bytes per second averaged over the configured interval	Flow stats are enabled by default, with a default interval of 10 seconds (see <xref>).
Node Stats Statistics related to a logical node. This is an aggregation of all flows that a node participates in.	• Total number of packets • Total number of bytes • Packets per second averaged over the configured interval • Bytes per second averaged over the configured interval	Node stats are enabled by default, with a default interval of 10 seconds (see <xref>).
Port Stats Ports Stats are per physical switch port	• Number of received packets • Number of transmitted packets • Number of received bytes • Number of transmitted bytes • Number of received packets per second averaged over the configured interval	Node stats are enabled by default, with a default interval of 30 seconds (see <xref>).

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Type	Fields	Default Behavior
	• Number of transmitted packets per second averaged over the configured interval • Number of received bytes per sec averaged over the configured interval • Number of transmitted bytes per sec averaged over the configured interval • Number of packets dropped by Rx • Number of packets dropped by Tx • Number of received errors • Number of collisions • Number of transmitted errors • Number of frame alignment errors • Number of packets with Rx overrun • Number of CRC errors	

Table 2.1: Supported statistics

REST API Methods

The Vello REST API uses GET, PUT, and DELETE methods as shown below.

Resource URI	GET Method	PUT Method	DELETE Method
A URI for a collection of data looks like this: `http://example.com/resources/`	List the URIs and perhaps other details of the collection's members. The collection in this example is called “resources”.	Replace the entire collection with another collection.	Delete the entire collection.
A URI for an individual element looks like this: `http://example.com/resources/item17`	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type. In this example, the collection is called “resources”, and the addressed member in the collection is “item 17”	Replace the addressed member of the collection, or create it if it doesn't exist..	

Table 2.2: Vello REST API methods

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

This page intentionally left blank.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

3: NX500 Configuration

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Accessing the REST API

You may read the NX500 Controller device configuration parameters using the REST API. Access the REST API using the IP address of the NX500 Controller and the REST Interface port. The basic syntax is:

```
https://<A.B.C.D>/<URI>
```

Where:

- `<A.B.C.D>` is the IP address of the NX500 Controller. By default, this is 192.168.1.1.
- `<URI>` is the URI to use. For example, to retrieve the NX 500 Controller IP configuration using cURL, enter the following on a command line prompt:

```
curl -k -u admin:vello123 https://192.168.1.1/
rest/v1.0/controller-config-ip/control-
plane{"mode":"static","ip-
addr":"192.168.1.1","sub-
net":"255.255.255.0","gateway":"0.0.0.0"}
```

See <xref> for a complete list of NX500 Controller cURL commands.

Default NX500 Controller Settings

The NX500 Controller shipped with the following default settings:

Configuration	Setting
Hostname	nvc
Date/Time	Unspecified
Logging	• Remote logging disabled • Set to “Information” (see the *User Guide*)
Configuration	**Setting**
Management IP	DHCP
Controlplane IP	192.168.1.1
HTTPS Access Port	443
Username (case sensitive)	admin
Password (case sensitive)	vello123

Table 3.1: NX500 Controller default settings

UBM Notes

UBM runs on the NX500 Controller and acts as a buffer between users and the REST API interface to handle client authentication and forward RESTful web API requests to the local REST server on your behalf.

UBM uses security files that exist on the NX500 Controller to establish secure HTTPS connections between clients and itself. These files are loaded during the manufacturing process to establish secure TLS connections to external entities, such as Vello VX1048/VX3048 switches and REST API clients.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

4: Vello REST API Methods

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

About the Vello REST API Methods

The Vello REST API is divided into three parts:

- **Node:** Together with the Flow APIs, these APIs define the logical network view.
- **Flow:** Together with the Node APIs, these APIs define the logical network view.
- **Topology/Device:** These APIs define the physical network view.

These APIs use a 'JSON Schema (see <xref>) as the format for passing messages between an application and the API.

API Descriptions

The API descriptions in this chapter use the following notation to explain each URI. The last column of each description table identifies a message format and whether that message is sent to the API by the application or is received by the application as a reply from the API. The message may also contain one or more error(s), which are listed by number and explained in *"REST API Error Codes" on page 35*.

/relevant-uri-part

HTTP Method Type (GET, PUT, or DELETE)	`~additional/uri/flow/with/:<identifier>`	Operation description.	• **Send:** JSON schema name that describes the data sent to the server • **Reply:** JSON schema name that describes the data received from the server • **Error code(s):** Listed as applicable

Example URI: `/rest/v1.0/relevant/uri/part/additional/uri/flow/with/<identifier>`

Table 4.1: REST API command listing example

Where: `<identifier>` is the specific ID of the flow, node, switch, etc.

Node REST APIs

A node is a logical collection of physical switch-ports that reside at the edge of a software defined network. Nodes can:

- have node-properties attached to them,
- provide statistics, and
- describe a flow end point.

Node

The **node** API allows you to add, edit, get, list, or delete one or more node objects. To use this API:

/node

DELETE `~/:<node ID>`	Delete a node object.	**Possible error(s):** 52, 102
GET `~`	Get a list of all node objects.	**Reply:** `<id:node-obj-list>`
GET `~/:<node ID>`	Get a node object.	• **Reply:** `<id:node-obj>` • **Possible error(s):** 52, 102
PUT `~/:<node ID>`	Create or modify a node object.	• **Send:** `<id:node-obj-create>` • **Possible error(s):** 51, 52, 53, 104

Example URI: `/rest/v1.0/node/<node ID>`

Where: `<node ID>` is the ID of the desired node.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Node Names

The **node names** API allows you to obtain a list of all node names. To use this API:

/node-names

GET `~`

Get a list of all node names.

Reply: `<id:name-list>`

Example URI: `/rest/v1.0/node-names`

Node Statistics

The **node stats** API allows you to obtain statistics for the specified node. To use this API:

/node-stats

GET `~/:<node ID>`

Get statistics for the specified node.

- **Reply:** `<id:node-stats>`
- **Possible error(s):** 52, 102

Example URI: `/rest/v1.0/node-stats/my-node`

Where: `<node ID>` is the ID of the desired node.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Flow REST APIs

A flow describes a connection between nodes. Data may flow between connected nodes, depending on the specified flow-property (policy rule). A unicast flow may have either unidirectional or bidirectional packet traffic. A multicast flow must have unidirectional packet traffic.

Flow

The **flow** API allows you to add, edit, get, list, or delete one or more flow objects. To use this API:

/flow

DELETE `~/:<flow ID>`	Delete a flow object.	**Possible errors:** 52, 101
GET `~`	Get a list of all flow objects.	**Reply:** **`<id:flow-obj-list>`**
GET `~/:<flow ID>`	Get a flow object.	• **Reply:** **`<id:flow-obj>`** • **Possible errors:** 52, 101
PUT `~/:<flow ID>`	Create or modify a flow object.	• **Send:** **`<id:flow-obj-create>`** • **Possible errors:** 51, 52, 53, 103, 105, 106

Example URI: `/rest/v1.0/node/<flow ID>`

Where: `<flow ID>` is the ID of the desired flow.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Flow Names

The **flow names** API allows you to obtain a list of all flow names. To use this API:

/flow-names

GET ~	Get a list of all flow names.	**Reply:** `<id:name-list>`

Example URI: `/rest/v1.0/flow-names`

Flow Destination

The **flow destination** API allows you to add or remove multicast target nodes. To use this API:

/flow-destination

DELETE `~/:<flow ID>/:node ID`	Remove a multicast target node from a flow.	**Possible errors:** 53, 101, 102, 150
PUT `~/:<flow ID>/:<node ID>`	Add a multicast target node to a flow. The node must have the same IP address as the other targets in the flow, and the flow must be a multi-cast flow.	**Possible errors:** 53, 101, 102, 150

Example URI: `/rest/v1.0/flow-destination/<flow ID>/<listener ID>`

Where:

- `<node ID>` is the ID of the desired node.
- `<flow ID>` is the ID of the desired flow.
- `<listener ID>` is the ID of the multicast listener.

Flow Statistics

The **flow stats** API allows you to obtain statistics for the specified flow. To use this API:

/flow-stats

GET `~/:<flow ID>`	Get statistics for the specified flow.	• **Reply:** `<id:flow-stats>` • **Possible error(s):** 52, 101, 200, 251
PUT `~/refresh/:time/:<flow ID>`	Set the sampling time for flow statistics for the specified flow.	**Possible errors:** 52, 101, 301

Example URI: `/rest/v1.0/flow-stats/refresh/8/<flow ID>`

Where: `<flow ID>` is the ID of the desired flow.

Flow Status

The **flow status** API allows you to obtain the current status of the specified flow. To use this API:

/flow-stats

GET `~/:<flow ID>`	Get the current status of the specified flow.	• **Reply:** `<id:path-stats-obj>` • **Possible errors:** 52, 101, 200, 251

Example URI: `/rest/v1.0/flow-status/<flow ID>`

Where: `<flow ID>` is the ID of the desired flow.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Flow Connection

The **flow connection** API allows you to obtain connection information for the specified flow(s). To use this API:

/flow-connection

GET `~`	Get connection information for all flows.	**Reply:** `<id:path-conn-all>`
GET `~/:<flow ID>`	Get connection information for the specified flow only.	• **Reply:** `<id:path-conn>` • **Possible error:** 100

Example URIs:
- `/rest/v1.0/flow-connection`
- `/rest/v1.0/flow-connection/<flow ID>`

Where: `<flow ID>` is the ID of the desired flow.

Topology/Device REST APIs

The topology/device APIs help you manage your physical network by providing information about the network and your ability to configure it. These APIs allow you to configure the NX500 Controller, obtain switch and port information, and configure SFLOW.

NX500 Controller IP Configuration

The **controller config ip** API allows you to query or configure the Management and Controlplane interfaces of the NX500 Controller. To use this API:

/controller-config-ip

GET `~/:management`	Get the Management interface configuration of the NX500 Controller.	**Reply:** `<id:config-ip>`
GET `~/:control-plane`	Get the Controlplane interface configuration of the NX500 Controller.	**Reply:** `<id:config-ip>`
PUT `~/:management`	Set the Management interface configuration of the NX500 Controller.	• **Send:** `<id:config-ip>` • **Possible errors:** 51, 53, 400
PUT `~/:control-plane`	Set the Controlplane interface configuration of the NX500 Controller.	• **Send:** `<id:config-ip>` • **Possible errors:** 51, 53, 400

Example URI: `/rest/v1.0/controller-config-ip/management`

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Switch

The **switch** API allows you to query the configuration of one or more VX1048/VX3048 switches. To use this API:

/switch

GET `~`	Get configuration information for all switches.	**Reply:** `<id:switch-array>`
GET `~/sw`	Get configuration information for the specified switch.	**Send:** `<id:switch>`

Example URI: `/rest/v1.0/switch`

Switch Names

The **switch names** API allows you to obtain a list of all switch names. To use this API:

/switch-names

GET `~`	Get a list of switch names.	**Reply:** `<id:name-list>`

Example URI: `/rest/v1.0/switch-names`

Switch Ports

The **switch names** API allows you to list all ports on either a specified switch or all switches. To use this API:

/switch-ports

GET `~`	Get all ports for all switches.	**Reply:** `<id:name-list>`
GET `~/sw:`	Get all ports for the specified switch.	• **Reply:** `<id:name-list>` • **Possible error:** 100

Example URIs:
- `/rest/v1.0/switch-ports`
- `/rest/v1.0/switch-ports/<switch ID>`

Where: `<switch ID>` is the ID of the desired switch.

Switch Connection

The **switch connection** API allows you to list all connections on either a specified switch or all switches. To use this API:

/switch-connection

GET `~`	Get all connections for all switches.	**Reply:** `<id:switch-conn-all>`
GET `~/sw:`	Get all connections for the specified switch.	• **Reply:** `<id:switch-conn>` • **Possible error:** 100

Example URIs:
- `/rest/v1.0/switch-connection`
- `/rest/v1.0/switch-connection/<switch ID>`

Where: `<switch ID>` is the ID of the desired switch.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Switch Port Statistics

The **switch port stats** API allows you to get statistics for the specified switch port. To use this API:

/switch-port-stats

GET `~/:<switch ID>/:<port ID>`	Get statistics for the specified switch port.	• **Reply:** **`<id:switch-port-stats>`** • **Possible errors:** 52, 200, 251

Example URIs: • `/rest/v1.0/switch-port-stats/<switch ID>/<port ID>`

Where:

- `<switch ID>` is the ID of the desired switch.
- `<port ID>` is the ID of the desired port.

REST API Error Codes

The REST API returns an error listing when an application error occurs in the REST interface. This listing consists of:

- Message body containing an error code
- Generic error message
- Auxiliary error message

For example:

```
{ "error-code": 0, "error-message": "No error",
"error-auxiliary": "This is just a test" }
```

The following table lists the possible error code values and provides a brief description for each code..

Error code	Description
0	No error occurred.
50	Invalid input was received.
51	An invalid input format was received.
52	An invalid identifier was received.
53	An invalid field value was received.
100	An entity did not exist.
101	A flow did not exist.
102	A node did not exist.
103	A flow-property did not exist.
104	A node-property did not exist.
105	A source node did not exist.
106	A target node did not exist.
107	Failed to change bandwidth.
150	An entity existed.
200	A resource was unavailable.
250	A generic read error occurred.
251	A bad read occurred.
300	A generic write error occurred.
301	A bad write occurred.
400	A generic failure occurred

Note: In addition to the errors described above, you may encounter HTTP or other computing errors that are not associated with the Vello REST API. Describing these errors is beyond the scope of this manual. These error codes are standardized, and you may research them online.

Table 4.2: REST API error codes

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

This page intentionally left blank.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

5: JSON

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

JSON Schema

The JSON schema used by the Vello REST API describes the format of JSON messages that are sent between the REST API and the application. The schema itself is written in JSON and consists of an array of schema objects. Each schema object describes the format for the schema element that it identifies. An element can be any logical or physical component of the network that you can modify programmatically, such as:

- Node (that you can create, monitor, modify, and delete)
- Flow (that you can create, monitor, modify, and delete)
- Port (physical aspect of the network that can be monitored but neither created nor deleted)

The following table shows selected examples of JSON instances of schema objects.

Schema Object (meta)	JSON Instance (actual)
`<id:name-list>`	`{ "items": [ "sw1", "sw2", "sw3" ] }`

`<id:node-obj-create>`	`{`
	`"node-element": [ "sw1-p1", "sw2-p22" ]`
	`"node-property-items": [ "my-node-prop" ]`
	`}`

`<id:flow-obj-create>`	`{`
	`"flow": {`
	`"source":"node-a",`
	`"destinations":["node-b"]`
	`},`
	`"flow-property": "my-flow-prop"`
	`}`

`<id:flow-direction>`	`"uni"`

Table 5.1: JSON schema object examples

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

JSON Common Objects

The JSON common objects defined in this section serve as components in the definitions of node, flow, and topology objects. These are the simplest-defined JSON objects in the API that form part of the building blocks for the more complex node, flow, and topology JSON object definitions described in:

- *"JSON Schema" on page 38*
- *"JSON Node Items" on page 42*
- *"JSON Flow Objects" on page 45*
- *"JSON Topology/Device Objects" on page 51*

IP Address

The **ip-address** JSON object describes an IPv4 IP address in the format <A.B.C.D>.

```
{
      "id": "ip-address",
      "type": "string",
      "format": "IPv4"
}
```

IP Mask

The **ip-mask** JSON object describes a netmask in CIDR format. For example, `192.168.100.0/24` represents the IP address 192.168.100.0 and its netmask of 255.255.255.0.

```
{
      "id": "ip-mask",
      "type": "number"
}
```

Switch Name

The **switch-name** JSON object describes a unique switch ID.

```
{
      "id": "switch-name",
      "type": "string",
      "format": "Vello switch name"
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Port Number

The **port-number** JSON object is a positive integer that represents a specific port on a switch.

```
{
      "id": "port-number",
      "type": "integer"
}
```

Switch Port Name

The **switch-port-name** JSON object refers to a specific port on a specific switch, such as `sw0001-p1` or `sw0002-p5`.

```
{
      "id": "switch-port-name",
      "type": "string"
}
```

Name

The **name** JSON object refers to a generic name string. The string may only contain alphanumeric characters (a-z, A-Z, 0-9), dashes (-), or underscores (_) and may not exceed 255 characters in length.

```
{
      "id": "name",
      "type": "string"
}
```

Name Array

The **name-array** JSON object refers to an array of names.

```
{
      "id": "name-array",
      "type": "array",
      "minItems": 0,
      "items": [ <id:name> ]
}
```

Name List

The **name-list** JSON object provides a name for an array of names.

```
{
      "id": "name-list",
      "type": "object",
      "properties": {
            "items": <id:name-array>
      }
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Port Array

The **port-array** JSON object refers to an array of ports.

```
{
      "id": "port-array",
      "type": "array",
      "minItems": 0,
      "items": [ <id:port-number> ]
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

JSON Node Items

These JSON objects are used for REST APIs that affect network nodes, such as calls that create, modify, configure, or delete nodes.

Node Element

The **node-element** JSON object refers to an array of switch-port name strings.

```
{
      "id": "node-element",
      "type": "array",
      "minItems": 1,
      "maxItems": 1,
      "items": [ <id:switch-port-name> ]
}
```

Node Status

The **node-status** JSON object refers to the current status of a node. The possible statuses are:

- Available
- Unavailable

```
{
      "id": "node-status",
      "type": "string"
}
```

Create Node Object

The **node-obj-create** JSON object creates or modifies a node with the following properties:

- **ID:** unique ID for the node
- **Node element:** Switch-port name string
- **Group name:** Group name for the node
- **IP address:** IP address of the node
- **IP Mask:** Netmask of the node
- **Description:** Text describing the node

Note: You may use a null value for any of these fields except `"id"` *and* `"node-element"`.

Multicasting is handled implicitly via the IP address. If the IP address is in the set 224.0.0.0-239.255.255.255 and is not one of the reserved multicast IP-addresses (224.0.0.0, 224.0.0.1, 224.0.0.2, 224.0.0.13), then the node is considered to be a multicast target node.

```
{
      "id": "node-obj-create",
      "type": "object",
      "properties": {
            "id": <id:name>,
            "node-element": <id:node-element>,
```

```
            "group-name": <id:name>,
            "ip-address": <id:ip-address>,
            "ip-mask": <id:ip-mask>,
            "description": <string>
      }
}
```

Node Object

The **node-obj** JSON object represents a node with the following information and having the following properties:

- **ID:** unique ID for the node
- **Node element:** Switch-port name string
- **Group name:** Group name for the node
- **IP address:** IP address of the node
- **IP Mask:** Netmask of the node
- **Description:** Text describing the node

Note: You may use a null value for any of these fields except `"id"` *and* `"node-element"`.

```
{
      "id": "node-obj",
      "type": "object",
      "properties": {
            "id": <id:name>,
            "node-element": <id:node-element>,
            "group-name": <id:name>,
            "ip-address": <id:ip-address>,
            "ip-mask": <id:ip-mask>,
            "description": <string>
      }
}
```

Node Object Array

The **node-obj-array** JSON object refers to an array of nodes.

```
{
      "id": "node-obj-array",
      "type": "array",
      "minItems": 0,
      "items": [ <id:node-obj> ]
}
```

List Node Objects

The **node-obj-list** JSON object provides a name for an array of nodes.

```
{
      "id": "node-obj-list",
      "type": "object",
      "properties": {
            "items": <id:node-obj-array>
      }
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Query Node

The **node-query** JSON object returns a list of node names that match your specified criteria. You may either specify one or more criteria or use one or more wildcard symbols (*) to return nodes with any value for the selected criteria, as follows:

- For the "group-name" and "ip-address" fields, the wildcard value is the string "".
- For the "ip-mask" field, this value is -1.

You may also use a null to indicate a wildcard value.

```
{
      "id": "node-query",
      "type": "object",
      "properties": {
            "group-name": <id:name>*,
            "ip-address": <id:ip-address>*,
            "ip-mask": <id:ip-mask>*
      }
}
```

Node Statistics

The **node-stats** JSON object is a list of the statistics for all of the flows that the specified node participates in.

```
{
      "id": "node-stats",
      "type": "object",
      "properties": {
            "items": [ <id:flow-stats> ]
      }
}
```

JSON Flow Objects

These JSON objects are used for REST APIs that affect network flows, such as calls that create, modify, configure, or delete flows.

Direction

The **flow-direction** JSON object is a string describing the flow direction. Valid values are:

- **uni:** unidirectional flow.
- **bi:** bidirectional flow.

```
{
      "id": "flow-direction",
      "type": "string"
}
```

Bandwidth

The **bandwidth** JSON object is a positive integer that defines the bandwidth for a flow in kilobytes per second (kbps). The system rounds the specified value to the nearest OpenFlow-supported value. OpenFlow values are typically powers of 10.

Note: A kilobyte is defined as 1,000 bytes and not 1,024.

```
{
      "id": "bandwidth",
      "type": "integer"
}
```

Hop Count

The **hop-count** JSON object is a positive integer that specifies the maximum number of hops for a flow. Specifying a maximum hop count that is too low for the physical network topology will cause the flow to fail.

```
{
      "id": "hop-count",
      "type": "integer"
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Flow Status

The **flow-status** JSON object refers to the current status of a flow. The possible statuses are:

- Available
- Unavailable
- Attempting

```
{
      "id": "flow-status",
      "type": "string"
}
```

Create Flow

The **flow-obj-create** JSON object creates or modifies a flow with the following properties:

- **ID:** unique ID for the flow
- **Source:** source node for the flow
- **Destinations:** Destination node(s) for the flow
- **Flow direction:** May be either `uni` (unidirectional) or `bi` (bidirectional)
- **Description:** Text describing the flow.
- **Maximum bandwidth:** Maximum bandwidth to reserve for this flow, in kilobytes per second (kbps). A value of 0 or null disables maximum bandwidth enforcement for this flow; the flow will be created with the default node port speed.
- **Maximum hop count:** Maximum number of hops from source node to destination node. Specifying a number that is too low for the physical network topology will cause the flow to fail.

Note: You may use a null value for the `"maximum-bandwidth"`, `"description"`, *and* `"maximum-hop-count"` *fields.*

```
{
      "id": "flow-obj-create",
      "type": "object",
      "properties": {
            "id": <id:name>,
            "source": <id:name>,
            "destinations": <id:name-array>,
            "flow-direction": <id:path-direction>,
            "description": <string>,
            "maximum-bandwidth": <id:bandwidth>,
            "maximum-hop-count": <id:hop-count>
      }
}
```

Flow Object

The **flow-obj** JSON object represents a flow with the following information and having the following properties:

- **ID:** unique ID for the node
- **Source:** Source node for the flow
- **Destination:** Destination node for the flow

- **Flow direction:** May be either `uni` (unidirectional) or `bi` (bidirectional)
- **Description:** Text describing the flow.
- **Maximum bandwidth:** Maximum bandwidth to reserve for this flow, in kilobytes per second (kbps). A value of 0 or null disables maximum bandwidth enforcement for this flow.
- **Maximum hop count:** Maximum number of hops from source node to destination node.

Note: You may use a null value for any of these fields.

```
{
      "id": "flow-obj",
      "type": "object",
      "properties": {
            "id": <id:name>,
            "source": <id:name>,
            "destinations": <id:name-array>,
            "flow-direction": <id:flow-direction>,
            "description": <string>,
            "maximum-bandwidth": <id:bandwidth>,
            "maximum- hop-count": <id:hop-count>
      }
}
```

Flow Object Array

The **flow-obj-array** JSON object refers to an array of flows.

```
{
      "id": "flow-obj-array",
      "type": "array",
      "minItems": 0,
      "items": [ <id:flow-obj> ]
}
```

Flow Object List

The **flow-obj-list** JSON object provides a name for an array of nodes.

```
{
      "id": "flow-obj-list",
      "type": "object",
      "properties": {
            "items": <id:flow-obj-array>
      }
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Flow Statistics

The **flow-stats** JSON object returns the following statistics for the specified flow:

- **ID:** Unique ID of the flow
- **Bytes:** Number of bytes handled by the flow
- **Bytes per second:** Speed of the flow in bytes per second
- **Packets:** Number of packets handled by the flow
- **Packets per second:** Speed of the flow in packets per second
- **Interval:** Duration in seconds
- **Description:** Text describing the flow

```
{
      "id": "flow-stats",
      "type": "object",
      "properties": {
            "id": <id:name>,
            "bytes": <number>,
            "bytes-per-sec": <number>,
            "packets": <number>,
            "packets-per-sec": <number>,
            "interval": <number>,
            "description": <string>
      }
}
```

Physical Path (Switch)

The **path-physical-switch** JSON object lists the path each flow takes from the source node to the destination node, as follows:

- **Switch:** Unique switch ID
- **Ingress port:** Port where flow data enters the specified switch
- **Egress port:** Port where flow data exits the specified switch

```
{
      "id": "path-physical-switch",
      "type": "object",
      "properties": {
            "switch": <id:name>,
            "ingress-port": <id:port-array>,
            "egress-port": <id:port-array>
      }
}
```

Physical Path (Switch Array)

The **path-physical-switch-array** JSON object refers to an array of **path-physical-switch** JSON objects.

```
{
      "id": "path-physical-switch-array",
      "type": "array",
      "minItems": 0,
      "items": [ <id:path-physical-switch> ]
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Physical Path

The **path-physical** JSON object refers to the number and names of the switches a flow travels through on its way from source node to destination node.

```
{
      “id”: “path-physical”,
      “type”: “object”,
      “properties”: {
            “count”: <number>,
            “path”: <id:path-physical-switch-
array>
      }
}
```

Physical Path Array

The **path-physical-array** JSON object refers to an array of **path-physical** JSON objects.

```
{
      “id”: “path-physical-array”,
      “type”: “array”,
      “minItems”: 0,
      “items”: [ <id:path-physical> ]
}
```

Flow Connection

The **flow-conn** JSON object describes an association between a path and the physical representation of that path.

```
{
      “id”: “flow-conn”,
      “type”: “object”,
      “properties”: {
            “id”: <id:name>,
            “physical-paths”: <id:path-physical-
array>
      }
}
```

Path Connection Array

The **path-conn-array** JSON object refers to an array of **path-conn** JSON objects.

```
{
      “id”: “path-conn-array”,
      “type”: “array”,
      “minItems”: 0,
      “items”: [ <id:flow-conn> ]
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

All Path Connections

The **path-conn-all** JSON object refers to the collection of all path-conn JSON objects.

```
{
      “id”: “path-conn-all”,
      “type”: “object”,
      “properties”: {
            “path-conn”: <id:path-conn-array>
      }
}
```

Path Destination Status

The **path-status-dest-obj** JSON object describes the status of a path through the specified destination node (switch and port).

```
{
      “id”: “path-status-dest-obj”,
       “type”: “object”,
      "properties": {
            "status":<string>,
            "switch":<string>,
            "port":<number>
      }
}
```

Path Destination Status Array

The **path-status-dest-obj-array** JSON object refers to an array of **path-status-dest-obj** JSON objects.

```
{
      "id": "path-status-dest-obj-array",
      "type": "array",
      "minItems":0,
      "items": [ <id:path-status-obj> ]
}
```

Path Status

The **path-status-obj** JSON object is the status of the specified flow.

```
{
       "id": "path-status-obj",
      "type": "object",
      "properties": {
            "id": <id:name>,
            "items": <id:path-status-dest-obj-
array>
      }
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

JSON Topology/Device Objects

These JSON objects are used to configure the physical network, including monitoring.

Configure SFLOW

The **sflow-config** JSON object provides the IP address and port for the SFLOW configuration.

```
{
      "id": "sflow-config",
      "type": "object",
      "properties": {
            "ip-address": <id:ip-address>,
            "port": <id:port-number>
      }
}
```

Port Type

The **port-type** JSON object refers to the type for the specified port. The possible values are:

- Copper
- Fiber
- Unavailable

```
{
      "id": "port-type",
      "type": "string"
}
```

Port State

The **port-state** JSON object indicates the status of the specified port. The possible values are:

- Up
- Down

```
{
      "id": "port-state",
      "type": "string"
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Port MAC Address

The **port-mac** JSON object refers to the MAC address of the specified port. A MAC address consists of six pairs of hexadecimal characters separated by a colon (:). For example: `12:45:89:AB:CD:EF`.

```
)
      "id": "port-mac",
      "type": "string"
}
```

Device Type

The **device-type** JSON object returns the device type. The possible device types are:

- **NX:** Vello NX500 Controller
- **VX:** Vello VX1048 or VX3048 switch
- **?X:** unknown

```
{
      "id": "device-type",
      "type": "string"
}
```

Switch Link

The **switch-link** JSON object defines a link between a port on a switch and another device based on the following properties:

- **Port:** Port number on the specified VX1048/VX3048 switch
- **Peer name:** Unique ID of the desired peer
- **Peer port:** Port number on the desired peer
- **Peer type:** Type of peer device

```
{
      "id": "switch-link",
      "type": "object",
      "properties": {
            "port": <id:port-number>,
            "peer-name": <id:switch-name>,
            "peer-port": <id:port-number>,
            "peer-type": <id:device-type>
      }
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Switch Link Array

The **switch-link-array** JSON object refers to an array of **switch-link** JSON objects.

```
{
      "id": "switch-link-array",
      "type": "array",
      "minItems": 0,
      "items": [ <id:switch-link> ]
}
```

Switch Inventory

The **switch-inv** JSON object returns the following information about the VX1048/VX3048 switches on the network:

- **Host name:** Host name of the switch
- **Dataplane ID:** Unique datapath ID assigned to the switch. Vello suggests using the switch MAC address as part of this ID.
- **Manufacturer:** Switch manufacturer (Vello)
- **Hardware:** Switch model (VX1048 or VX3048)
- **Software:** VelIOS firmware version that the switch is running
- **Dataplane Description:** Human-readable description of the Dataplane ID
- **Serial number:** Unique serial number of each switch

```
{
      "id": "switch-inv",
      "type": "object",
      "properties": {
            "host-name": <string>,
            "dataplane-id": <number>,
            "description-mfr": <string>,
            "description-hw": <string>,
            "description-sw": <string>,
            "description-dataplane": <string>,
            "serial-number": <string>
      }
}
```

Switch

The **switch** JSON object returns the following information about the specified switch:

- **ID:** unique switch ID
- **Inventory:** Switch inventory information (host name, manufacturer, hardware, software, dataplane, and serial number)
- **IP configuration:** IP address configuration

```
{
      "id": "switch",
      "type": "object",
      "properties": {
            "id": <string>,
            "inventory": <switch-inv>,
            "ip-config": <config-ip>,
      }
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Switch Array

The **switch-array** JSON object refers to an array of **switch** JSON objects.

```
{
      "id": "switch-array",
      "type": "array",
      "minItems": 0,
      "items": [ <id:switch> ]
}
```

Switch Connection

The **switch-conn** JSON object refers to the association between the specified switch and the device(s) that switch is connected to.

```
{
      "id": "switch-conn",
      "type": "object",
      "properties": {
            "name": <id:switch-name>,
            "type": <id:device-type>,
            "links": <id:switch-link-array>
      }
}
```

Switch Connection Array

The **switch-conn-array** JSON object refers to an array of **switch-conn** JSON objects.

```
{
      "id": "switch-conn-array",
      "type": "array",
      "minItems": 0,
      "items": [ <id:switch-conn> ]
}
```

All Switch Connections

The **switch-conn-all** JSON object returns a collection of all switch connections.

```
{
      "id": "switch-conn-all",
      "type": "object",
      "properties": {
            "switch-conn": <id:switch-conn-array>
      }
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Switch Port

The **switch-port** JSON object returns the following properties for the specified port on the specified switch:

- **Number:** Port number
- **Current capacity:** Current capacity of the physical port in kilobytes per second (kbps)
- **Max capacity:** Maximum capacity of the physical port in kilobytes per second (kbps)
- **Type:** Port type (can be copper, fiber, or unavailable)
- **State:** Port state (can be up or down)
- **MAC address:** Mac address of the port, which consists of six pairs of hexadecimal characters; for example, 04:5A:CB:FA:95:23

```
{
      "id": "switch-port",
      "type": "object",
      "properties": {
            "number": <number>,
            "current-capacity": <number>,
            "max-capacity": <number>,
            "type": <id:port-type>,
            "state": <id:port-state>,
            "mac": <id:port-mac>
      }
}
```

Switch Ports

The **switch-ports** JSON object lists the ports available on the specified switch.

```
{
      "id": "switch-ports",
      "type": "object",
      "properties": {
            "name": <name>,
            "count": <number>,
            "ports": <id:switch-port-array>
      }
}
```

Switch Ports Array

The **switch-ports-array** JSON object refers to an array of **switch-ports** JSON objects.

```
{
      "id": "switch-ports-array",
      "type": "array",
      "minItems": 0,
      "items": [ <id:switch-ports> ]
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

List Switch Ports

The **list-switch-ports** JSON object represents the collection of switch-ports-array objects.

```
{
      "id": "switch-ports-list",
      "type": "object",
      "properties": {
            "switch-ports": <id:switch-ports-
array>
      }
}
```

Switch Port Statistics

The **switch-port-stats** JSON object returns the following statistics for the specified port on the specified switch:

- **ID:** Unique ID of the specified port
- **TX bytes:** Total number of bytes transmitted through the port
- **TX bytes per second:** Number of bytes transmitted through the port per second
- **TX packets:** Total number of packets transmitted through the port
- **TX packets per second:** Number of packets transmitted through the port per second
- **RX bytes:** Total number of bytes received through the port
- **RX bytes per second:** Number of bytes received through the port per second
- **RX packets:** Total number of packets received through the port
- **RX packets per second:** Number of packets received through the port per second
- **Interval:** Duration in seconds
- **Description:** Human-readable string describing the statistics (may be a null value)

```
{
      "id": "switch-port-stats",
      "type": "object",
      "properties": {
            "id": <id: switch-port>,
            "tx-bytes": <number>,
```

```
            "tx-bytes-per-sec": <number>,
            "tx-packets": <number>,
            "tx-packets-per-sec": <number>,
            "rx-bytes": <number>,
            "rx-bytes-per-sec": <number>,
            "rx-packets": <number>,
            "rx-packets-per-sec": <number>,
            "interval": <number>,
            "description": <string>,
      }
}
```

IP Configuration Mode

The **config-ip-mode** JSON object represents the IP configuration mode, which can be either static or DHCP.

```
{
      "id": "config-ip-mode",
      "type": "string"
}
```

IP Configuration

The **config-ip** JSON object is the collection of information related to IP configuration.

- **Mode:** IP address mode (static or DHCP)
- **IP address:** IP address
- **Subnet:** Subnet mask
- **Gateway:** Gateway IP address

```
{
      "id": "config-ip",
      "type": "object",
      "properties": {
            "mode": <id:config-ip-mode>,
            "ip-addr": <id:ip-address>,
            "subnet": <id:ip-address>,
            "gateway": <id:ip-address>
      }
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

This page intentionally left blank.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

6: cURL Examples

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Accessing cURL

cURL is a command-line tool that you run either from the command prompt (Windows) or the Terminal (OSX or Linux) to either send data to a server or receive data from a server. The process of opening a command line depends on your operating system. You may then enter cURL commands. The following sections list some examples of using cURL with the Vello REST API.

Note: You may download cURL from http://curl.haxx.se for Windows, Mac OSX, Linux, or Ubuntu.

Windows

To access a command line in Windows:

1. Click **Start>Run** to open the **Run** dialog.
2. Enter `cmd` and then press [ENTER].

Mac OSX

To access a command line in Mac OSX:

1. Navigate to the **Applications\Utilities** folder.
2. Open the Terminal application.

Linux/Ununtu

To access a command line in Linux or Ubuntu:

- **Linux:** open a bash prompt or Terminal window.
- **Ubuntu:** Open a bash prompt by navigating to **Applications>Accessories>Terminal**.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

cURL Node Examples

This section contains examples of node operations using cURL.

Creating Nodes

This command creates the node `1st node` that is defined by all of objects in the instance. In this example, as part of its definition, the node is located on port 1 of switch SW0000001E0800039E, has the net mask 32, and the IP address 10.0.0.1.

```
curl  -k -u admin:vello123 -X PUT https://
192.168.1.1/rest/v1.0/node/n1 -d '{"node-ele-
ment": ["SW0000001E0800039E-p1"], "id": "node",
"group-name": "BLAH", "ip-mask": 32, "descrip-
tion": "1st node", "ip-address": "10.0.0.1"}'
```

This command creates the node `2nd node`.

```
curl  -k -u admin:vello123 -X PUT https://
192.168.1.1/rest/v1.0/node/n2  -d '{"node-ele-
ment": ["SW0000001E0800039E-p2"], "id": "node",
"group-name": "BLAH", "ip-mask": 32, "descrip-
tion": "2nd node", "ip-address": "10.0.0.2"}'
```

Get Node Information

This command returns information for the node `n1`.

```
curl  -k -u admin:vello123 -X GET https://
192.168.1.1/rest/v1.0/node/n1
```

Delete a Node

This command deletes node n1. The system deletes the specified node without prompting you to confirm the deletion. Delete nodes with care.

```
curl  -k -u admin:vello123 -X DELETE https://
192.168.1.1/rest/v1.0/node/n1
```

Get All Nodes

This command returns a list of attributes for all nodes in the network.

```
curl  -k -u admin:vello123 -X GET https://
192.168.1.1/rest/v1.0/node
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Create a Flow

This commend create a unidirectional flow `flow 1` from node `n1` to node `n2` with a maximum bandwidth of 600 and no maximum hop count or description specified.

```
curl   -k -u admin:vello123 -X PUT https://
192.168.1.1/rest/v1.0/flow/flow1 -d '{"destina-
tions":["n2"],"source":"n1","flow-direc-
tion":"uni","description":"FLOW","maximum-
bandwidth":600,"maximum-hop-count":null}'
```

Create a Flow Destination

This command defines a destination node for a multicast flow.

```
curl   -k -u admin:vello123 -X PUT https://
192.168.1.1/rest/v1.0/flow-destination/
MCAST_FLOW/l2
```

Configure Static Refresh Rate

This command reconfigures the statistic refresh rate of the flow `my_flow`.

```
curl   -k -u admin:vello123 -X PUT  https://
192.168.1.1/rest/v1.0/flow-stats/refresh/15/
my_flow
```

Configure NX500 Management IP

The following commands reconfigure the NX500 Controller management IP address. The first command defines a static IP address.

```
curl  -k -u admin:vello123 -X PUT https://
192.168.1.1/rest/v1.0/controller-config-ip/man-
agement -d '{"mode":"static","ip-
addr":"192.168.1.1","subnet":"255.255.0.0","gate-
way":"192.168.254.1"}'
```

This command defines a DHCP IP address.

```
curl  -k -u admin:vello123 -X PUT https://
192.168.1.1/rest/v1.0/controller-config-ip/man-
agement -d '{"mode":"dhcp","ip-
addr":"0.0.0.0","subnet":"0.0.0.0","gate-
way":"0.0.0.0"}'
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

7: Python Scripts

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Using Python with the Vello REST API

The Vello NX/VX documentation package includes configurable and executable Python code that uses the REST API. You must have a Python interpreter installed on your computer to work with Python code. You may download a Python interpreter from http://www.python.org/download/. The Python website also contains documentation and tutorials at http://www.python.org/doc/.

The available Python scripts allow you to:

- Work with nodes (see next section)
- Work with flows (see *"Flow Operations" on page 66*)
- Enable, view, and disable statistics (see *"Statistics" on page 70*)

The Vello REST API Python scripts use the following open-source libraries (available online):

- **httplib:** This library contains the HTTP methods (GET, PUT, POST, DELETE) for making HTTP requests to the REST API.
- **JSON:** This library is required for processing the resulting JavaScript Object Notation (JSON) response.

Node Operations

You may use Python to perform the following node operations with the REST API:

- *"Add Node" on page 64*
- *"Show Single Node" on page 65*
- *"List All Nodes" on page 65*
- *"Delete Single Node" on page 66*
- *"Delete All Nodes" on page 66*

Add Node

The **node add** function adds a node. To use this function:

```
node add --id=<node ID> --switchport=<switchport>
--ipaddr=<A.B.C.D> --ipmask=<yyy.yyy.yyy.yyy> --
desc=<descstring>
```

Where:

- `<node ID>` is the unique ID for the node you are creating.
- `<switchport>` is the switch DPID & port number information (e.g SW0000001E0800039E-p1) where you want to add your node>.
- `<A.B.C.D>` is the properly formatted IP address for the new node.
- `<yyy.yyy.yyy.yyy>` is the netmask for the new node.
- `<descstring>` is a free-text entry (up to 255 haracters) describing the node.

The output from this operation is:

- **On success:** `Node successfully added`
- **On failure:** `Node failed to be added`
-

Show Single Node

The **node show** function displays detailed information for the selected node. To use this function:

`node show --id=<node ID>`

Where:

`<node ID>` is the unique ID of the node you are viewing.

The output from this operation is:

- **On success:**

```
Node successfully retrieved
Node ID: <node ID>
Switchport: <switchport>
IP/Mask: <IP>/<mask>
Desc: <descstring>
```

- **On failure:** `Node failed to be retrieved`

Where:

- `<node ID>` is unique ID of the selected node.
- `<switchport>` is the specific VX1048/VX3048 switch and port.
- `<IP>` is the IP address of the specified port.
- `<mask>` is the netmask of the specified port.
- `<descstring>` is free text describing the node.

List All Nodes

The **node list** function displays either summary or detailed information for all nodes. To use this function:

- **`node list`:** Lists summary information for all nodes.
- **`node list --detail`:** Lists detailed information for all nodes.

The output for this operation is:

- **On success (summary):**

```
Node list successfully retrieved
Found <#> nodes:
Node ID: <node 1 ID>
...
Node ID: <node n ID>
```

- **On success (detail):**

```
Node list successfully retrieved
Found <#> nodes:
Node ID: <node 1 ID> (Switchport: <switchport>,
IP: <IP>/<mask>, Desc: <descstring>)
...
Node ID: <node n ID (Switchport: <switchport>,
IP: <IP>/<mask>, Desc: <descstring>)
```

- **On failure:** `Node list failed to be retrieved`

Where:

- `<node _ ID>` is the unique ID of each node.
- `<switchport>` is the specific VX1048/VX3048 switch and port.
- `<IP>` is the IP address of the specified port.
- `<netmask>` is the netmask of the specified port.
- `<descstring>` is free text describing the node.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Delete Single Node

The **node delete** function deletes the specified node. To use this function:

```
node delete --id=<node ID>
```

Where:

`<node ID>` is the unique ID of the node you are deleting.

The output from this operation is:

- **On success:** `Node successfully deleted`
- **On failure:** `Node failed to be deleted`
-

Delete All Nodes

The **node delete-all** function deletes all nodes. To use this function:

```
node delete-all
```

The output from this operation is:

```
Node <node 1 ID> successfully deleted
...
Node <node n ID> successfully deleted
```

Where:

`<node _ ID>` is the unique ID of each node being deleted.

Flow Operations

You may use Python to perform the following flow operations with the REST API:

Add a Flow

The **flow add** function adds a flow. To use this function:

```
flow add --id=<flow ID> --node1=<node 1 ID> --
node2=<node 2 ID> --direction=<uni|bi> [--band-
width=<bw>] [--maxhops=<hops>]
```

Where:

- `<flow ID>` is the unique ID to assign to this flow.
- `<node 1 ID>` is the Vello VX1048/VX3048 switch where the flow will originate (source).

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

- `<node 2 ID>` is the Vello VX1048/VX3048 switch where the flow will terminate (destination).
- `<uni|bi>` specifies whether the flow is unidirectional (from node 1 to node 2) or bidirectional.
  - `uni` specifies a unidirectional flow.
  - `bi` specifies a bidirectional flow.
- `<bw>` is the bandwidth reservation for this flow, in kbps. IK is defined as 1,000 bytes, not 1,024.
- `<hops>` is the maximum number of hops this flow may use between source and destination nodes. Specifying a number that is too low for the available network topology will cause the flow to fail.

The output from this operation is:

- **On success:** **`Flow successfully added`**
- **On failure:** **`Flow failed to be added`**

Add Multicast Destination Node

The **flow add-node** function adds a multicast destination node. To use this function:

```
flow add-node --id=<flow ID> --node1=<node ID>
```

Where:

- `<flow ID>` is the unique ID of the flow that the destination will receive (see *"Add a Flow" on page 66* and *"List all Flows" on page 69*).
- `<node ID>` is the unique ID of the multicast node you are creating.

The output from this operation is:

- **On success:** **`Multicast flow destination successfully added`**
- **On failure:** **`Multicast flow destination failed to be added`**

Display Flow Status

The **flow status** function displays the status of the specified flow. To use this function:

```
flow status --id=<flow ID>
```

Where:

`<flow ID>` is the unique ID of the flow you are viewing.

The output from this operation is:

```
Flow ID: <flow ID>
node: (Switchport: <switchport> Status: <status string>)
```

Where:

- `<switchport>` is the specific VX1048/VX3048 switch and port.
- `<status string>` is the current flow status.

Display Flow Trace

The **flow connection** function provides either a summary or detailed trace of the path taken by the specified flow To use this function:

```
flow connection --id=<flow ID>
```

Where:

`<flow ID>` is the unique ID of the flow you are tracing.

The output from this operation is:

```
Flow ID: <flow ID>
Count: <number of switch flows>
Switch: <first switch> Ingress-port: [<port #>]
Egress-port: [<port #>]
...
Switch: <last switch> Ingress-port: [<port #>]
Egress-port: [<port #>]
```

Where:

- <____ switch> is the ID of each switch on the flow.
- <port #> is the specific port number where data entered (ingress) and exited (egress) on each switch.

Show a Flow

The **flow show** function displays either summary or detailed information for the selected flow. To use this function:

- **flow show --id=<flow ID>:** Lists summary information for the selected flow.
- **flow show --id=<flow ID> --detail:** Lists detailed information for the selected flow.

Where:

<flow ID> is the unique ID of the selected flow.

The output from this operation is:

- **On success (summary):**

```
Flow successfully retrieved
Flow ID: <flow ID>
node1: <node 1 ID>
node2: <node 2 ID>
Direction: <uni|bi>
```

- **On success (detail):**

```
Flow successfully retrieved
Flow ID: <flow ID>
node1: <node 1 ID> (Switchport: <switchport>,
IP: <IP>/<mask>, Desc: <descstring>)
node2: <node 2 ID> (Switchport: <switchport>,
IP: <IP>/<mask>, Desc: <descstring>)
Direction: <uni|bi>
```

- **On failure:** `Flow failed to be retrieved`
- <flow ID> is the unique ID to assign to this flow.
- <node 1 ID> is the Vello VX1048/VX3048 switch where the flow will originate (source).
- <node 2 ID> is the Vello VX1048/VX3048 switch where the flow will terminate (destination).
- <uni|bi> specifies whether the flow is unidirectional (from node 1 to node 2) or bidirectional.
  - uni specifies a unidirectional flow.
  - bi specifies a bidirectional flow.
- <bw> is the bandwidth reservation for this flow, in kbps. IK is defined as 1,000 bytes, not 1,024.
- <hops> is the maximum number of hops this flow may use between source and destination nodes. Specifying a number that is too low for the available network topology will cause the flow to fail.

List all Flows

The **flow list** function displays either summary or detailed information for all flows. To use this function:

- **`flow list`:** Lists summary information for all flows.
- **`flow list --detail`:** Lists detailed information for all flows.

The output for this operation is:

- **On success (summary):**

```
Flow list successfully retrieved
Found <#> flows:
Flow ID: <flow 1 ID>
...
Flow ID: <flow n ID>
```

- **On success (detail):**

```
Flow list successfully retrieved
Found <#> flows:
Flow ID: <flow 1 ID>
node1: <node1 ID> (Switchport: <switch and
port>, IP: <IP>/<mask>, Desc: <descstring>)
node2: <node2 ID> (Switchport: <switch and
port>, IP: <IP>/<mask>, Desc: <descstring>)
Direction <uni|bi>
...
Flow ID: <flow n ID)
node1: <node1 ID> (Switchport: <switch and
port>, IP: <IP>/<mask>, Desc: <descstring>)
node2: <node2 ID> (Switchport: <switch and
port>, IP: <IP>/<mask>, Desc: <descstring>)
Direction <uni|bi>
```

- **On failure:** `Flow list failed to be retrieved`

Where:

- `<#>` is the number of flows found.
- `<flow _ ID>` is the unique ID of each flow.
- `<node_ ID>` is unique ID of each Vello VX1048/VX3048 switch on each flow.
- `<switchport>` is the specific VX1048/VX3048 switch and port.
- `<IP>` is the IP address of the specified port.
- `<mask>` is the netmask of the specified port.
- `<descstring>` is a free text entry (up to 255 characters) describing the node.
- `<uni|bi>` is the flow direction (unidirectional or bidirectional, respectively).

List Switches

The **switch list** function lists each of the VX1048/VX3048 switches found. To use this function:

```
switch list
```

The output from this operation is:

```
Found <#> switches:
Switch: <first switch name>
...
Switch: <last switch name>
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Delete a Flow

The **flow delete** function deletes the specified flow. To use this function:

```
flow delete --id=<flow ID>
```

Where:

`<flow ID>` is the unique ID of the flow you are deleting.

The output from this operation is:

- **On success:** **`Flow successfully removed`**
- **On failure:** **`Flow failed to be removed`**

Delete Multicast Destination Node from a Flow

The **flow delete-node** function deletes a multicast destination node from a flow. To use this function:

```
flow delete-node --id=<flow ID> --node1=<node ID>
```

Where:

- `<flow ID>` is the unique ID of the flow that the destination is receiving (see *"Add a Flow" on page 66* and *"List all Flows" on page 69*).
- `<node ID>` is the unique ID of the node being deleted.

The output from this operation is:

- **On success:** **`Multicast flow destination successfully deleted`**
- **On failure:** **`Multicast flow destination failed to be deleted`**

Delete all Flows

The **flow delete-all** function deletes all flows. To use this function:

```
flow delete-all
```

The output from this operation is:

- **On success:**

  `Flow <flow 1 ID> successfully deleted`
  `...`
  `Flow <flow n ID> successfully deleted`

- **On failure:** **`Multicast flow destination failed to be deleted`**

Where:

`<flow _ ID>` is the unique ID of each flow.

Statistics

You may use Python to perform the following flow operations with the REST API:

- *"Enable/Disable Flow Statistics" on page 71*
- *"Enable/Disable Flow Statistics" on page 71*
- *"Show Port Statistics" on page 71*
- *"Show Flow Statistics" on page 71*
- *"Show Node Statistics" on page 72*

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

Enable/Disable Flow Statistics

The **stats flow** function enables or disables statistics reporting for the selected path. To use this function:

- **`stats flow enable --id=<path ID> --interval=<time>`:** Enables statistics reporting for the selected flow.
- **`stats flow disable --id=<path ID>`:** Disables statistics reporting for the selected flow.

Where:

`<path ID>` is the unique ID of the selected path.

The output from this operation is:

- **On success (enable):** **`Statistics for connection successfully set to refresh interval <time>`**
- **On success (disable):** **`Statistics for connection successfully disabled`**
- **On error:** **`Failed to set connection statistics`**

Where

`<time>` is the time interval in seconds.

Show Port Statistics

The **stats port show** function displays statistics for the selected port. To use this function:

`stats port show --switchport=<switchport>`

Where:

`<switchport>` is the specific VX1048/VX308 switch and port.

The output from this operation is:

- **On success:**

```
Port <switchport> statistics:
TX packets: <value> bytes: <value>
RX packets: <value> bytes: <value>
Rates (averaged over <interval> seconds):
TX packets/sec: <value> bytes/sec: <value>
RX packets/sec: <value> bytes/sec: <value>
```

- **On error:** **`Failed to retrieve port statistics`**

Show Flow Statistics

The **stats flow show** function displays statistics for the selected connection. To use this function:

- **`stats flow show --id=<path ID>`:** Shows the flow for the selected path ID.
- **`stats flow show --node1=<node 1 ID> --node2=<node 2 ID> --direction=<uni|bi>`:** Shows the flow between the selected notes.

Where:

- `<path ID>` is the unique ID of the selected path.
- `<node _ ID>` is the unique ID of the selected node.
- `<uni|bi>` is the flow direction (unidirectional or bidirectional, respectively).

The output from this operation is:

- **On success:**

```
Connection <flow ID> statistics:
Packets: <value> Bytes: <value>
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

```
Rates (averaged over <interval> seconds):
Packets/sec: <value> Bytes/sec: <value>
```

- **On error:** `Failed to retrieve connection statistics`

Show Node Statistics

The **stats node show** function lists all connections to the selected client/product and prints the statistics for each connection. To use this function:

```
stats node show --id=<node ID>
```

Where:

`<node ID>` is the unique ID of the selected path.

The output from this operation is:

- **On success:**

```
Connection <first <flow ID> statistics:
Packets: <value> Bytes: <value>
Rates (averaged over <interval> seconds):
Packets/sec: <value> Bytes/sec: <value>
...
Connection <last <flow ID> statistics:
Packets: <value> Bytes: <value>
Rates (averaged over <interval> seconds):
Packets/sec: <value> Bytes/sec: <value>
```

- **On error:** `Failed to retrieve node statistics`

Sample Python Scripts

The following Python scripts provide a basic example of querying the statistics for a particular path. The following sample scripts are split into two files:

- common.py handles the connection setup for all REST API calls
- path_stats.py demonstrates a request for path statistics and how to access the response fields.

Note: You may download these sample scripts from Vello.

common.py

The **common.py** script handles connection setup for REST API calls:

```
#!/usr/bin/env python
import httplib
import base64
# login credentials
https_username = "admin"
https_password = "vello123"
# set these to point to the controller
controller_addr = '172.18.89.138'
controller_port = 443
headers = {}
headers['Content-Type'] = 'application/json'
headers['Content-Length'] = 0
headers['User-Agent'] = 'Vello REST API script'
headers['Accept'] = '*/*'
def get_http_code(code):
      return httplib.responses[code]
def transact(method, uri, body):
      conn = httplib.HTTPSConnec-
tion(controller_addr, controller_port)
      conn.connect()
         request = conn.putrequest(method, uri)
      headers['Content-Length'] = len(body)
      auth = base64.encodestring("%s:%s" %
(https_username, https_password)).strip()
      headers['Authorization'] = "Basic %s" % auth
      for k in headers:
            conn.putheader(k, headers[k])
      conn.endheaders()
      conn.send(body)
      resp = conn.getresponse()
      resp_read = resp.read()
      conn.close()
      return (resp.status, resp.reason,
resp_read)
```

path_stats.py

The **path_stats.py** script gets path statistics:

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

```
#!/usr/bin/env python
import sys
import common
import json
# validate the arguments
if (len(sys.argv) != 2):
      print "Incorrect number of parameters."
      print "Usage: " + sys.argv[0] + " <flow ID>"
          exit(0)
# set up the request method and URI
req_method = "GET"
req_str = "/rest/v1.0/flow-stats/" + sys.argv[1]
# perform the transaction and get the response
resp = common.transact(req_method, req_str, "")
code_str = common.get_http_code(resp[0])
status_str = str(resp[0]) + " " + code_str
print "Status: "
print status_str
print ""
# Raw output
print "Response: "
print resp[2]
print ""
# Generic JSON output with formatting
print "JSON: "
json_reply = json.loads(resp[2])
print json.dumps(json_reply, sort_keys=True,
indent=4)
print ""
```

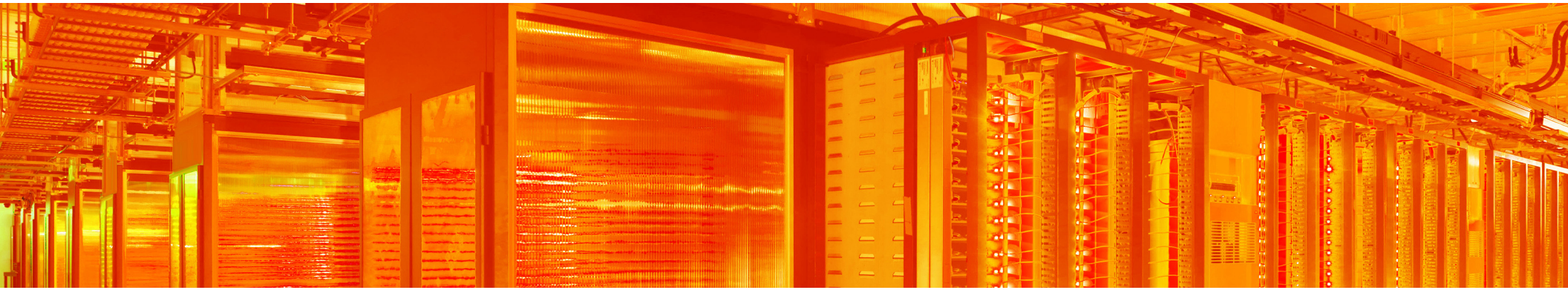
This operation returns the following output:

```
Status:
200 OK
Response:
{
      "id": "mpls",    "bytes": 903607386,
"bytes-per-sec":0,       "packets": 7059432,
"packets-per-sec": 0,    "interval": 10,
"description": ""
      }
JSON:
{
      "bytes": 903607386,
      "bytes-per-sec": 0,
      "description": "",
      "id": "mpls",
      "interval": 10,
      "packets": 7059432,
      "packets-per-sec": 0
}
```

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com

This page intentionally left blank.

AUTHORED BY ANTHONY HERNANDEZ - (415)786-2081 - anthony94122@outlook.com



NX500, VX1048, VX3048 REST API Guide, v1.00 (06/2013)

Contact Information:

Vello Systems, Inc.
1530 O'Brien Drive
Menlo Park, CA 94025
Phone: 650-324-7600
Fax: 650-324-7601
Toll-free: 1-866-MY-GIGES (1-866-694-4437)
Email: support@vellosystems.com
Website: www.vellosystems.com