ipinfusion™

DANOS-Vyatta edition

Disaggregated Network Operating System Version 2009a

Scripting Reference Guide

October 2020

Contents

Chapter 1. Copyright Statement

IP Infusion Inc.
3965 Freedom Circle, Suite 200
Santa Clara, CA 95054
+1 408-400-1900

http://www.ipinfusion.com/.

For support, questions, or comments via E-mail, contact:

support@ipinfusion.com.

Chapter 2. Preface

Document conventions

The document conventions describe text formatting conventions, command syntax conventions, and important notice formats used in this document.

Notes, cautions, and warnings

Notes, cautions, and warning statements may be used in this document. They are listed in the order of increasing severity of potential

hazards.

Note: A Note provides a tip, guidance, or advice, emphasizes important information, or provides a reference to related information.

Attention: An Attention statement indicates a stronger note, for example, to alert you when traffic might be interrupted or the device might reboot.

CAUTION: A Caution statement alerts you to situations that can be potentially hazardous to you or cause damage to hardware, firmware, software, or data.

DANGER: A Danger statement indicates conditions or situations that can be potentially lethal or extremely hazardous to you. Safety labels are also attached directly to products to warn of these conditions or situations.

Text formatting conventions

Text formatting conventions such as boldface, italic, or Courier font are used to highlight specific words or phrases.

Format	Description
bold text	Identifies command names. Identifies keywords and operands.
italic text	Identifies emphasis. Identifies variables. Identifies document titles.
`Courier font`	Identifies CLI output. Identifies command syntax examples.

Command syntax conventions

Bold and italic text identify command syntax components. Delimiters and operators define groupings of parameters and their logical relationships.

Convention	Description
bold text	Identifies command names, keywords, and command options.
italic text	Identifies a variable.
[ ]	Syntax components displayed within square brackets are optional. Default responses to system prompts are enclosed in square brackets.
{ **x** | **y** | **z** }	A choice of required parameters is enclosed in curly brackets separated by vertical bars. You must select one of the options.
x | **y**	A vertical bar separates mutually exclusive elements.
< >	Nonprinting characters, for example, passwords, are enclosed in angle brackets.
...	Repeat the previous element, for example, *member*[*member*...].
\	Indicates a “soft” line break in command examples. If a backslash separates two lines of a command input, enter the entire command at the prompt without the backslash.

Chapter 3. About This Guide

This guide describes how to use the Scripting API (also called the configd API) to programmatically configure and administer DANOS-Vyatta edition.

This guide also describes how to create vcli scripts to access commands.

Chapter 4. Overview

The Router Scripting API

The router Scripting API, or configd API, lets you programmatically configure and manage the router through configd, a YANG-based data-modeling management daemon.

Note: Configuring a router by using the Scripting API is very similar to configuring the device by using the CLI because the CLI itself uses this API to configure and manage the router.

The Scripting API lets you perform two categories of actions on routers: configuration and operation.

The structure of the data that is used by t Scripting API is defined in the YANG models and varies freely from the configd API.

Supported languages

The DANOS-Vyatta edition Scripting API is available in the following languages:

- Perl
- Python
- Ruby

Note: Many examples in this guide use the Python API.

Path formats

A path in Perl, Python, and Ruby is represented as either a space-separated string or a native-list object. The following method calls specify the same path.

```
set("foo bar baz")
set(["foo", "bar", "baz"])
```

You can use the path encoding that is more appropriate for the particular context from which the method is currently being called.

Database names

The Scripting API supports the following three parameter database options in API calls.

Table 1. Parameter databases

Parameter database	Description
AUTO	Automatically selects the appropriate database (RUNNING or CANDIDATE), depending on the context. Operational mode: RUNNING. Configuration mode: CANDIDATE.
RUNNING	Stores the committed state of the system.
CANDIDATE	Stores the configuration information for the current configuration session. If an API call is made externally to a session, the Scripting API reverts to the RUNNING database.

Chapter 5. Using the Scripting API

Note: Most examples in the following sections are in Python, but the usage is similar in other languages.

Setting up a connection to configd

When using the Scripting API to write a script, you must first set up a connection to configd by importing the configd module, which is included in the Vyatta package, and creating a `configd.client` object. You can then communicate with configd by using this object.

To set up a connection to configd, perform the following steps.

Table 2. Setting up a connection to configd

Step	Command
Import the configd module.	`from vyatta import configd`
Set up the connection to configd by instantiating a client object.	`client = configd.Client()`

Note: If there is a problem connecting to configd, the API raises the `configd.FatalException` exception.

Setting up a configuration session

Before you can run configuration commands from your script, you must set up a configuration session with configd.

To set up a configuration session, use the `client.session_setup()` function, which takes as input a session ID, as shown in the following Python example.

```
client.session_setup(str(os.getpid()))
```

The session ID must be a unique string. A common practice is to use the process ID (PID).

Note: If you initialize the client from inside a configuration session, the client inherits the session ID from the environment. To specify a different session ID, create a new session, as shown in the preceding example

.

Manipulating the configuration data on the router

After setting up a configuration session, you can manipulate the configuration data on the router.

The most common manipulation methods are the following. For more information about the supported methods, refer to configd API Methods. Most of these methods take as input a configuration path, which is represented as a space-separated string or sequence of strings.

Method	Returns	Description
`get(path)`	List of strings	Provides access to subtrees of the configuration database.
`tree_get_dict(path)`	Dictionary	Provides access to a subtree of a configuration database.
`node_exists(Database, path)`	Boolean	Reports whether a given path exists in the requested database.
`validate_path(path)`	String	Determines whether a path is valid according to the schema.
`set(path)`	String	Creates a new path in the configuration data store.
`delete(path)`	String	Removes a given path from the configuration data store.
`discard()`	String	Throws away all pending (uncommitted) changes for the current session.
`validate()`	String	Checks that the candidate configuration meets all the constraints that are modeled in the schema.
`commit(string)`	String	Validates and applies the candidate configuration.

Note: If a `configd.Exception` is thrown by these methods, a string that contains the informational data about the exception is returned by the router. You can safely ignore most return values, but it is a good practice to review the informational data in case it contains information that can help you resolve errors.

If an error occurs when calling these methods, configd exceptions are thrown, as shown in the following Python example. These errors include providing invalid paths to methods.

```
>>> client.set("foo bar")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/dist-packages/vyatta/configd.py", line 298, in
 set
    return _configd.Client_set(self, *args)
vyatta.configd.Exception: Configuration path: foo bar [foo] is not valid
```

The following sample Python script, add_bridge.py, shows how to manipulate configuration data.

```
#!/usr/bin/env python
from vyatta import configd
import os, sys
client = configd.Client()
client.session_setup(str(os.getpid()))
try:
    br1_path = "interfaces bridge br1".split(" ")
    client.set(br1_path + ["description", "bridge to nowhere"])
```

```
    client.set(br1_path + "address 1.1.1.1/32".split(" "))
    print(client.tree_get_dict(br1_path))
    client.validate()
    client.commit("add bridge to nowhere")
    client.session_teardown()
except configd.Exception as e:
    sys.stderr.write(e.what())
    exit(1)
```

Running this script twice gives you a better feel for how the script behaves. On the first run, everything is as expected.

```
vyatta@vyatta# python add_bridge.py
{u'aging': 300, u'tagnode': u'br1', u'description': u'bridge to nowhere',
 u'address': [u'1.1.1.1/32']}
```

However, on the second run, an exception is thrown because the path already exists in the configuration tree.

```
vyatta@vyatta# python add_bridge.py
Configuration path: interfaces bridge br1 description [bridge to nowhere]
 is not valid
Node exists
```

You can handle individual errors independently or you can check for conditions before making API calls, as shown in the following example.

```
#!/usr/bin/env python
from vyatta import configd
import os, sys
client = configd.Client()
client.session_setup(str(os.getpid()))
br1_path = "interfaces bridge br1".split(" ")
try:
    client.set(br1_path + ["description", "bridge to nowhere"])
except configd.Exception as e:
    print e
br1_address = br1_path + "address 1.1.1.1/32".split(" ")
if not client.node_exists(client.AUTO, br1_address):
    client.set(br1_address)
print(client.tree_get_dict(br1_path))
try:
    client.validate()
    client.commit("add bridge to nowhere")
    client.session_teardown()
except configd.Exception as e:
    sys.stderr.write(e.what())
    client.session_teardown()
    exit(1)
```

Running commands in operational mode

Note: Most data about the operational state on the router does not have a YANG model. As a result, you cannot use the Scripting API to call operational commands. Instead, you can use remote procedure calls (RPCs), as described in Using RPCs to run operational commands.

Operational data in the YANG data-modeling language is encoded as nodes in a read-only data tree. The data encoded in the tree varies freely from the configuration. The data represents the runtime state of the system. The operational and configuration trees are returned together when both types of nodes exist within the modeled hierarchy.

The data tree can be requested from configd by using the `tree_get_full` family of methods. Each method lets you select the encoding of the returned data.

Python uses a **dict** encoding that returns a dictionary representing the configuration hierarchy.

Perl and Ruby use a **hash** encoding that returns the native object for these languages.

The following is a list of the `tree_get_full` family of methods.

- tree_get_full
- tree_get_full_xml
- tree_get_internal
- tree_get_full_encoding

Note: The `tree_get_full_dict()` method is the native Python interface for retrieving data trees and uses some syntactic sugar. This method does not require the `database` parameter and it is the method that you will most likely use when accessing data trees. The other methods are provided in case you need to use a raw-text encoding. You can also select the dictionary encoding, which can be one of the following two JavaScript Object Notation (JSON) encodings. The default encoding is JSON.

Table 3. Supported JSON encodings

Encoding	Description
JSON (Recommended)	Defined by the RESTCONF specification, encodes lists as a list of objects with the entries keys represented as elements of the objects.
INTERNAL	Encodes lists as a dictionary with the entries keys represented as the keys to the dictionary. This encoding is specific to the router.

The following are sample Python `tree_get_full` calls.

```
>>> client.tree_get_full(client.AUTO, "hypervisor vm-state")
'{"vm-state":{"vm":[{"name":"vm0","state":"running"},{"name":"vm1","state":
"running"}]}}'

>>> client.tree_get_full_xml(client.AUTO, "hypervisor vm-state")
```

```
'<data><vm-state xmlns="urn:vyatta.com:mgmt:vyatta-hypervisor:1"><vm
 xmlns="urn:vyatta.com:mgmt:vyatta-hypervisor:1"><name
 xmlns="urn:vyatta.com:mgmt:vyatta-hypervisor:1">vm0</name><state
 xmlns="urn:vyatta.com:mgmt:vyatta-hypervisor:1">running</state></vm><vm
 xmlns="urn:vyatta.com:mgmt:vyatta-hypervisor:1"><name
 xmlns="urn:vyatta.com:mgmt:vyatta-hypervisor:1">vm1</name><state
 xmlns="urn:vyatta.com:mgmt:vyatta-hypervisor:1">running</state></vm></vm-s
tate></data>'

>>> client.tree_get_full_internal(client.AUTO, "hypervisor vm-state")
'{"vm-state":{"vm":{"vm0":{"state":"running"},"vm1":{"state":"running"}}}}'

>>> client.tree_get_full_dict("hypervisor vm-state")
{u'vm-state': {u'vm': [{u'state': u'running', u'name': u'vm0'}, {u'state':
 u'running', u'name': u'vm1'}]}}
 [provide paths if comf. With def. no need ot specify db or encoding]
>>> client.tree_get_full_dict("hypervisor vm-state",
 database=client.CANDIDATE, encoding=client.INTERNAL_ENCODING)
{u'vm-state': {u'vm': {u'vm0': {u'state': u'running'}, u'vm1': {u'state':
 u'running'}}}}
```

Using RPCs to run operational commands

You can use RPCs, which are independent of the Scripting API, to call the operational mode daemon, which returns the output of any router show command.

Note: RPCs access individual services. For any service, RPCs must conform to the YANG specifications for that service.

For each supported RPC, the YANG specifications define an input object and an output object. To make RPC calls, use the native `call_rpc` methods that are supported by your language of choice (`call_rpc_dict` for Python and `call_rpc_hash` for Perl and Ruby).

The following example shows how to make an RPC call in Python to restart the router.

```
 >>>
client.call_rpc_dict("vyatta-hypervisor-v1", "restart-vm",
{"name":"vm0"})
 {}
```

Like the `tree_get_full` family of method calls that are defined in the Scripting API, the `call_rpc` family of calls contains a method that lets you pass raw strings back and forth without having to convert them to objects that are native to the host language.

The command RPC

The `vyatta-opd:command` RPC is defined as follows:

```
rpc command {
```

```
        configd:call-rpc "oprpc";
        input {
                leaf command {
                        type enumeration {
                                enum show;
                        }
                        default show;
                }
                leaf args {
                        type string;
                }
        }
        output {
                leaf output {
                        type string;
                }
        }
}
```

RPC Calls

The following is a sample Python RPC call.

```
>>> client.call_rpc_dict("vyatta-opd-v1", "command", {"command":"show",
 "args":"interfaces"})
{u'output': u'Codes: S - State, L - Link, u - Up, D - Down, A -
 Admin Down\nInterface        IP Address                        S/L
 Description\n---------        ----------                         ---
 -----------\nbr0              192.168.1.1/24                     u/u
 \ndp0p0s20f0       172.22.20.142/24                u/u  \ndp0p0s20f1
      -                                 A/D  \ndp0p0s20f2       -
                       A/D  \ndp0p0s20f3       -
           A/D  \ndp0vhost0        -                                  u/u
 \ndp0vhost1        -                                  A/D  \n'}
```

To format the output of the preceding example so that the columns are aligned, use the print command with the RPC call, as shown in the following example.

```
>>> print client.call_rpc_dict("vyatta-opd-v1", "command",
 {"command":"show", "args":"interfaces"})["output"]
Codes: S - State, L - Link, u - Up, D - Down, A - Admin Down
Interface        IP Address                        S/L  Description
---------        ----------                        ---  -----------
br0              192.168.1.1/24                    u/u
dp0p0s20f0       172.22.20.142/24                  u/u
dp0p0s20f1       -                                 A/D
dp0p0s20f2       -                                 A/D
dp0p0s20f3       -                                 A/D
dp0vhost0        -                                 u/u
dp0vhost1        -                                 A/D
```

Scripting examples

Setting data plane interface addresses

The Perl, Python, and Ruby examples in this section show how to configure the data plane interfaces to receive their IP addresses from a DHCP server. The equivalent router command follows:

set interfaces dataplane <*interface*> address dhcp

Perl scripting

```
#!/usr/bin/env perl

use strict;
use warnings;
use lib '/opt/vyatta/share/perl5';
use Vyatta::Configd;

my $client = Vyatta::Configd::Client->new();

sub setup_interface_address {
    my ($intf_name) = @_;
    print("$intf_name:");
    my @addr = $client->get("interfaces dataplane $intf_name address");
    printf("%s\n", join(", ", @addr));
    $client->set("interfaces dataplane $intf_name address dhcp")
 if (scalar(@addr) == 0);
}

$client->session_setup("$$");
eval {
    map { setup_interface_address($_) } $client->get("interfaces
 dataplane");
    $client->commit("setup interface addresses");
} || warn "$@\n";
$client->session_teardown();
```

Python scripting

```
import vyatta.configd as configd
from sys import stderr, stdout
from os import getpid

def setup_interface_address(client, intf_name):
    stdout.write(intf_name + ":")
    path = ["interfaces", "dataplane", intf_name, "address"]
    addrs = client.get(path)
```

```
    print(", ".join(addrs))
    if len(addrs) == 0 :
        client.set(path + ["dhcp"])


def main():
    client = configd.Client()
    client.session_setup(str(getpid()))
    try:
        for intf_name in client.get("interfaces dataplane"):
            setup_interface_address(client, intf_name)
        client.commit("setup interface addresses")
    except configd.Exception as e:
        stderr.write(str(e))
    client.session_teardown()


if __name__ == "__main__":
    main()
```

Ruby scripting

```
require "vyatta/configd";

def setup_interface_address(client, intf_name)
  print(intf_name, ":")
  path = ["interfaces", "dataplane", intf_name, "address"]
  addrs = client.get(path)
  puts addrs.join(", ")
  if addrs.length == 0
    client.set(path << "dhcp")
  end
end


if __FILE__ == $PROGRAM_NAME
  client = client = Vyatta::Configd::Client.new
  client.session_setup($$.to_s())
  begin
    client.get("interfaces dataplane").each { |name|
 setup_interface_address(client, name) }
    client.commit("setup interface addresses")
  rescue Vyatta::Configd::Exception => e
    puts e.message
  end
  client.session_teardown()
end
```

Monitoring the host

The following Perl script periodically pings a remote address from a router. If the ping loss is beyond a certain percentage, the script prints a message to the standard error stream (STDERR).

```
#!/usr/bin/env perl

use strict;
use warnings;

use lib "/opt/vyatta/share/perl5";

use Getopt::Long;
use Try::Tiny;
use Vyatta::Configd;

# To run this as a background task :
# systemd-run --unit=monitor-8.8.8.8 --user -- ./monitor-host --address
 8.8.8.8

# The background task can be managed using systemctl
# $ systemctl --user status monitor-8.8.8.8
# ● monitor-8.8.8.8.service - /home/vyatta/./monitor-host --address 8.8.8.8
# Loaded: loaded
 (/home/vyatta/.config/systemd/user/monitor-8.8.8.8.service; static)
#    Drop-In: /home/vyatta/.config/systemd/user/monitor-8.8.8.8.service.d
#     └─50-Description.conf, 50-ExecStart.conf
#  Active: active (running) since Fri 2015-09-18 14:37:52 UTC; 29s ago
#    Main PID: 4952 (perl)
#
 CGroup: /
user.slice/user-1000.slice/user@1000.service/monitor-8.8.8.8.service
#               └─4952 perl /home/vyatta/./monitor-host --address 8.8.8.8
#
# $ systemctl --user stop monitor-8.8.8.8

# All output from a script run as above can be viewed with journalctl or
 syslog
# for instance:
# Sep 18 14:55:15 vyatta monitor-host[5181]: packet loss to host 1.1.1.1
 exceeded 50%

sub address_reachable {
    my ( $client, $address, $count, $percentage ) = @_;
    try {
        my $result = $client->call_rpc_hash( "vyatta-op-v1", "ping",
            { "host" => $address, "count" => $count } );
        return ( $result->{"rx-packet-count"} / $count ) * 100 >=
 $percentage;
    }
    catch {
        print STDERR "$_\n";
```

```
        return;
    };
}

sub usage {
    print STDERR "uasge $0:\n";
    print STDERR " --address     the address to ping [required]\n";
    print STDERR " --count       the number of pings to send [default:
 10]\n";
    print STDERR " --percentage  the tolerable amount of loss [default:
 50]\n";
    exit(1);
}

my ( $address, $count, $percentage );

GetOptions(
    "address=s"    => \$address,
    "count=s"      => \$count,
    "percentage=s" => \$percentage,
) or usage();

usage() unless defined($address);
$count      = 10 unless defined($count);
$percentage = 50 unless defined($percentage);

my $client = Vyatta::Configd::Client->new();

for ( ; ; ) {
    if ( !address_reachable( $client, $address, $count, $percentage ) ) {
        printf STDERR "packet loss to host %s exceeded %s%%\n", $address,
          $percentage;
    }
}
```

Language-specific conventions

Perl

All Perl methods that take a path as input may take either a space-separated string or an arrayref structure representing the path.

The return strings of Scripting API Perl methods are as arrayref structures or values returned on the stack, depending on the return context

Table 4. Return context

Context	Value type
scalar	arrayref
array	values on stack

The following four special methods provide convenience sugar for the Perl API.

```
get($path, $database) #database is optional $AUTO if not defined
tree_get_hash($path, $opts) # opts are optional: { "encoding" =>
 $JSON_ENCODING, "database" => $AUTO }
tree_get_full_hash($path, $opts)  # opts: { "encoding" => $JSON_ENCODING,
 "database" => $AUTO }
call_rpc_hash($ns, $name, $input) #input is a hash representation of the
 input stanza of the RPC definition
```

The Perl configd module exports the following constants so they may be used more easily.

```
$AUTO
$CANDIDATE
$RUNNING
$EFFECTIVE
$CHANGED
$UNCHANGED
$ADDED
$DELETED
$LEAF
$MULTI
$TAG
$CONTAINER
```

These constants can be imported in the standard way.

```
use Vyatta::Configd qw($AUTO $CANDIDATE $RUNNING);
```

Python

All Python methods that take a path as input may take either a space-separated string or an arrayref structure representing the path.

The Scripting API Python methods return strings as dictionary structures.

The following four special methods provide sugar for Python.

```
get(self, path, database=AUTO)
tree_get_dict(self, path, database=AUTO, encoding=JSON_ENCODING)
tree_get_full_dict(self, path, database=AUTO, encoding=JSON_ENCODING)
call_rpc_dict(self, ns, name, input)
```

Ruby

All Ruby methods that take a path as input may take either a space-separated string or an arrayref structure representing the path.

```
get(path) -> string
tree_get_hash(path) -> hash
tree_get_full_hash(path) -> hash
call_rpc_hash(ns, name, input) -> hash
```

Chapter 6. configd API Methods

Types

Database

An enumeration that specifies the supported database parameters.

Declaration

```
enum Database { AUTO, RUNNING, CANDIDATE }
```

NodeStatus

An enumeration that specifies the supported node status values.

Declaration

```
enum NodeStatus { UNCHANGED, CHANGED, ADDED, DELETED }
```

NodeType

An enumeration that specifies the supported node types.

Declaration

```
enum NodeType { LEAF, MULTI, CONTAINER, TAG }
```

Methods

call_rpc

Calls an RPC.

Declaration

Note: RPCs that are called by this function must be defined in the YANG data models.

```
call_rpc(ns, name, input)
```

Parameters

ns

XML namespace for the model in which the RPC is defined.

name

Nname of an RPC to call.

input

JSON-encoded definition of the input schema.

Returns

`call_rpc()` returns a string containing the JSON-encoded output schema that is defined in the RPC.

call_rpc_xml

Calls an RPC.

Declaration

Note: RPCs that are called by this function must be defined in the YANG data models.

```
call_rpc_xml(ns, name, input)
```

Parameters

ns

XML namespace for the model in which the RPC is defined.

name

Name of an RPC to call.

input

XML-encoded definition of the input schema.

Returns

`call_rpc_xml()` returns a string containing the XML-encoded output schema that is defined in the RPC.

commit

Validates and applies the candidate configuration. The candidate configuration is applied to the router resulting in the running configuration being updated if the transaction is successful.

Declaration

```
commit(comment)
```

Parameters

comment

Comment that describes changes made during this commit operation. An empty string is allowed.

Returns

`commit()` returns a string that contains all the informational messages that were generated during the commit operation.

delete

Removes a path from the configuration data store. The client has to be attached to a configuration session for this call to work. If the error occurs, that error is thrown as an exception. Deleting a path that does not exist is considered an error.

Declaration

```
delete(path)
```

Parameters

path

Path to a configuration node that is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`delete()` returns a string that contains all the informational messages that were generated during the delete operation.

discard

Throws away all pending (uncommitted) changes for this session.

Declaration

```
discard()
```

get_features

Gets a map of schema IDs and their corresponding enabled features.

Declaration

```
get_features(void)
```

Returns

`get_features()` returns a map of schema IDs and their corresponding enabled features.

get_help

Gets help about the children of a node.

Declaration

```
get_help(path, from_schema)
```

Parameters

path

Path of a parent node for which help is requested. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

from_schema

Boolean value. If true, generates the help from the schema definition and the data tree. If false, help is retrieved only from the data tree.

Returns

Returns a map that contains a list of the children of a node and their help strings.

load

Replaces the router configuration the candidate database with the configuration from a file.

Declaration

```
load(file)
```

Parameters

file

Path to a file that contains the new configuration.

Returns

`load()` returns `true` if the load operation is successful. Otherwise, it returns `false`.

node_is_default

Reports whether a path is the default path in a database.

Declaration

```
node_is_default(db, path)
```

Parameters

db

Database to query.

path

Path to a configuration node. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`node_is_default()` returns `true` if the path is the default path in the database. Otherwise, it returns `false`.

---

node_exists

Reports whether a path exists in a database.

Declaration

```
node_exists(db, path)
```

Parameters

db

Database to query.

path

Path to a configuration node. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`node_exists()` returns `true` if the path exists in the database. Otherwise, it returns `false`.

---

node_get

Queries a database for the values at a path.

Declaration

```
node_get(db, path)
```

Parameters

db

Database to query.

path

Path to a configuration node. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`node_get()` returns `true` if the path is the default path in the database. Otherwise, it returns `false`.

node_get_status

Reports the status of a node in a configuration tree.

Declaration

```
node_get_status(db, path)
```

Parameters

db

Database to query.

path

Path to a configuration node. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`node_get_status()` returns the status of a node. For more information about the possible status values of a node, refer to NodeStatus.

save

Saves the currently running configuration to the saved configuration.

Declaration

```
save()
```

session_attach

Attaches the client to a session ID. If the session does not exist, an exception is raised.

Declaration

```
session_attach (sessid)
```

Parameters

sessid

Session ID.

session_changed

Reports whether the current session has configuration changes.

Declaration

```
session_changed()
```

Returns

`session_changed()` returns `true` if the session has configuration changes. Otherwise, it returns `false`.

session_exists

Reports whether the current session still exists.

Declaration

Note: This function determines whether a shared session has been torn down by another instance.

```
session_exists()
```

Returns

`session_exists()` returns `true` if the current session still exists. Otherwise, it returns `false`.

session_lock

Attempts to lock the current session. If the session is currently locked, an exception is raised.

Declaration

```
session_lock()
```

session_locked

Reports whether the current configuration session is locked. A lock can be set to prevent multiple writers on a shared session. A lock also prevents changes during a commit operation. The lock of a session is released if the client that took the lock disconnects.

Declaration

```
session_locked()
```

Returns

`session_locked()` returns `true` if the current session is locked. Otherwise, it returns `false`.

session_saved

Reports whether the current configuration session has been saved to the running configuration.

Declaration

```
session_saved()
```

Returns

`session_saved()` returns `true` if the current configuration session has been saved to the running configuration. Otherwise, it returns `false`.

session_setup

Creates a new configuration session and makes that session the context for this instance of the Client object.

Declaration

Note: Commonly, the PID of the process is used as the session ID, but the ID can be any arbitrary string.

```
session_setup(sessid)
```

Parameters

sessid

Session ID.

Returns

`session_setup()` returns none.

session_teardown

Destroys a configuration session.

Declaration

Note: A session must be destroyed as soon as it is no longer required because configd maintains the state indefinitely, unless configd is instructed to destroy the session. However, in some contexts, the destruction of a session by the client might not be appropriate. Therefore, exercise caution when using this command.

```
session_teardown()
```

Returns

`session_teardown()` returns none.

session_unlock

Attempts to unlock the current session. If the session is currently not locked by this process, an exception is raised.

Declaration

```
session_unlock()
```

Returns

`session_unlock()` returns none.

set

Creates a new path in the configuration data store. The client has to be attached to a configuration session for this call to work. If an error occurs, the error is thrown as an exception. Creating a path that already exists is considered an error.

Declaration

```
set(path)
```

Parameters

path

Path to a configuration node. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`set()` returns a string that contains all the informational messages that were generated during the set operation.

template_get_allowed

Runs the configd:allowed extension to get the help values for a value node in the schema tree.

Declaration

```
template_get_allowed(path)
```

Parameters

path

Path in a schema tree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`template_get_allowed()` returns a string that contains the help values for a path in a schema tree.

template_get_children

Accesses the children of the schema node at a path in a schema tree.

Declaration

```
template_get_children(path)
```

Parameters

path

Path in a schema tree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`template_get_children()` returns a string that contains the children at a path in a schema tree.

template_validate_path

Determines whether a path is valid according to a schema.

Declaration

```
template_validate_path(path)
```

Parameters

path

Path in a schema tree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`template_validate_path()` returns `true` if a path is valid according to a schema. Otherwise, it returns `false`.

template_validate_values

Determines whether a path is valid according to a schema and validates all values according to the syntax of the schema.

Declaration

```
template_validate_values(path)
```

Parameters

path

Path in a schema tree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`template_validate_values()` returns `true` if a path is valid according to a schema. Otherwise, it returns `false`.

tree_get

Provides access to a subtree of a configuration database. This call is equivalent to calling `tree_get_encoding()` and specifying JSON_ENCODING as the encoding to use for the returned string.

Declaration

```
tree_get(db, path)
```

Parameters

db

Database from which to get a subtree.

path

Configuration path at which to root the subtree. The path is represented as either a space-separated string or an array of strings that represent the elements of the path.

Returns

`tree_get()` returns a JSON-encoded string representing a subtree at a specific location.

tree_get_encoding

Retrieves a configuration tree by using an encoding.

Declaration

```
tree_get_encoding(db, path, encoding)
```

Parameters

db

Database from which to get a subtree.

path

Path to a configuration node at which to root the tree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

encoding

Encoding of the returned string.

Returns

`tree_get_encoding()` returns a string in an encoding that represents a tree at a specific location.

tree_get_full

Provides access to a subtree of the operational data store. The operational data store consists of the configuration database and any data about the modeled operational state. This call is equivalent to calling `tree_get_full_encoding()` and specifying JSON_ENCODING as the encoding to use for the returned string.

Declaration

```
tree_get_full(db, path)
```

Parameters

db

Database from which to get a subtree.

path

Path to a configuration node at which to root the subtree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`tree_get_full()` returns a JSON-encoded string that represents a subtree at a specific location.

tree_get_full_encoding

Provides access to a subtree of the operational data store. The operational data store consists of the configuration database and any data about the modeled operational state. These trees are encoded in the specified encoding and returned as a string.

Declaration

```
tree_get_full_encoding(db, path, encoding)
```

Parameters

db

Database from which to get a subtree.

path

Path to a configuration node at which to root the subtree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

encoding

Encoding of the returned string.

Returns

`tree_get_full_encoding()` returns a string in an encoding that represents a tree at the given location.

tree_get_full_internal

Provides access to a subtree of the operational data store. The operational data store consists of the configuration database and any data about the modeled operational state. This call is equivalent to calling `tree_get_full_encoding()` and specifying INTERNAL_ENCODING as the encoding to use for the returned string.

Declaration

```
tree_get_full_internal(db, path)
```

Parameters

db

Database from which to get a tree.

path

Path to a configuration node at which to root the subtree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`tree_get_full_internal()` returns a string in the INTERNAL_ENCODING encoding that represents a tree at the given location.

tree_get_full_xml

Provides access to a subtree of the operational data store. The operational data store consists of the configuration database and any data about the modeled operational state. This call is equivalent to calling `tree_get_full_encoding()` and specifying XML_ENCODING as the encoding to use for the returned string.

Declaration

```
tree_get_full_xml(db, path)
```

Parameters

db

Database from which to get a subtree.

path

Path to a configuration node at which to root the subtree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`tree_get_full_xml()` returns an XML-encoded string that represents a subtree at the given location.

tree_get_internal

Provides access to a subtree of the configuration database. This call is equivalent to calling `tree_get_encoding()` and specifying INTERNAL_ENCODING as the encoding to use for the returned string.

Declaration

```
tree_get_internal(db, path)
```

Parameters

db

Database from which to get a subtree.

path

Path to a configuration node at which to root the subtree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

encoding

Encoding of the returned string.

Returns

`tree_get_internal()` returns a string in the INTERNAL_ENCODING encoding that represents a tree at the given location.

tree_get_xml

Provides access to a subtree of the configuration database. This call is equivalent to calling `tree_get_encoding()` and specifying XML_ENCODING as the encoding to use for the returned string.

Declaration

```
tree_get_xml(db, path)
```

Parameters

db

Database from which to get a subtree.

path

Path to a configuration node at which to root the subtree. The path is represented as either a space-separated string or an array of strings, which, in turn, represent the elements of the path.

Returns

`tree_get_xml()` returns an XML-encoded string that represents a subtree at the given location.

validate

Checks that the candidate configuration meets all constraints that are modeled in the schema.

Declaration

```
validate()
```

validate_path

Checks that a path can be set.

Declaration

```
validate_path(path)
```

Parameters

path

The path to the configuration node. The path is represented as either a space-separated string or an array of strings that represent the elements of the path.

Returns

`validate_path()` returns all informational messages that were generated during the validation process. If the path is invalid, this function throws an exception.

Chapter 7. Automatically running scripts on startup

You can instruct the router to run a script on startup. The script runs as part of the Linux service that handles system configuration. The script runs after the router applies the system configuration, which means that the script can modify the loaded configuration.

To instruct the router to automatically run a script on startup, perform the following steps.

1. Add the script to the /config/scripts folder.
   For example, enter the following command to create a script in the /config/scripts folder and add the `touch /myfile` command to it. The script creates an empty file in the root folder.

   ```
   echo 'touch /myfile' >> /config/scripts/my-postconfig-bootup.script
   ```

   Note: The script name can be any name and does not have to end with *.script*.
2. Make the script executable.
   For example, enter the following command to make my-postconfig-bootup.script executable.

   ```
   sudo chmod 770 /config/scripts/my-postconfig-bootup.script
   ```
3. Use your favorite editor to add a line to /config/scripts/vyatta-postconfig-bootup.script to run the script.

   Note: The router runs the scripts referenced in /config/scripts/vyatta-postconfig-bootup.script in the order in which they appear.

   For example, enter the following command to wto run /config/scripts/vyatta-postconfig-bootup.script on restart.

   ```
   echo '/config/scripts/my-postconfig-bootup.script'
    >> /config/scripts/vyatta-postconfig-bootup.script
   ```

Chapter 8. Using vcli

vcli shell scripting interface

The vcli shell scripting interface provides a special wrapper to the bash shell, which allows you to seamlessly access CLI commands on the router. This vcli shell operates as if it is in the router configuration mode, but you have to set up and terminate sessions before manipulating the candidate data tree.

Invoking vcli

To invoke vcli, enter the following command:

vcli [*options*]

For more information about supported vcli options, enter the following command:

```
$ vcli -h
Invalid option: -h
vcli [ OPTIONS ]
        OPTIONS: { -s SID | -c COMMAND | -i | -f FILE | -- SCRIPT_OPTIONS }
                -i interactive modeless shell
                -s SID configuration session id if not provided uses PID
                -c COMMAND one shot command
                -f FILE file to run
        NOTES:
                '-f FILE' is treated as a delimiter for SCRIPT_OPTIONS as
 well
                vcli will read a full script from standard in if no options
 are provided
```

Specifying a session ID

When invoking vcli, you can use the **-s** option to specify the session ID to use. By default, if this option is not provided, vcli uses its process ID (PID). The **-s** option is useful for connecting to an existing session, such as invoking a script from an existing configuration session or for debugging a NETCONF transaction.

Establishing a persistent configuration session

If you are not connecting vcli to an existing session between vcli and a router, then, to establish a persistent configuration session with the router to manipulate candidate configuration, you must use the **configure** command in the script before entering the

configuration commands. The **configure** command lets your script enter the Configuration mode on the router.

Because the session is persistent, if you do not perform session cleanup before exiting the vcli script, the session persists until the router is restarted. To prevent persistence until router restart, perform session cleanup by using the **end_configure** command at the end of the configuration section of your script.

The following example shows how to use the **configure** and **end_configure** commands to establish a persistent configuration session and then perform session cleanup before exiting the script.

```
#!/bin/vcli -f
configure
# Add configuration commands here
...
end_configure
```

Configuration manipulation

The following CLI commands are available in vcli and work exactly as they do in the Vyatta CLI.

commit

delete

edit

load

save

set

show

top

up

validate

Note: All vcli commands require full configuration paths. The shortest unambiguous-match abbreviations do not work because vcli scripts are run noninteractively.

Convenience commands

In addition to the standard CLI commands, vcli provides the following convenience commands.

- **list** *path*

Takes a path and returns its elements in a space-separated list. This command allows you to programmatically traverse the configuration tree. Following are examples.

```
vyatta@vyatta# vcli -c 'list interfaces'
bridge dataplane loopback

vyatta@vyatta# vcli -c 'list interfaces dataplane'
dp0p0s20f0

vyatta@vyatta# vcli -c 'list interfaces dataplane dp0p0s20f0'
address ip ipv6 mtu
```

- **interactive** and **noninteractive**

  These commands control the way operational commands prompt a user. When you enter the **noninteractive** command, any subsequent operational mode commands do not prompt for input, but they do accept all default values. The **interactive** command reverts to the normal mode of operation and prompts for input.

Entering operational commands

To invoke operational commands by using vcli, enter the **run** command, as shown in the following example.

```
vyatta@vyatta# vcli -c 'run show interfaces'
Codes: S - State, L - Link, u - Up, D - Down, A - Admin Down
Interface        IP Address                        S/L  Description
---------        ----------                        ---  -----------
br0              192.168.1.1/24                    u/u
dp0p0s20f0       172.22.20.142/24                  u/u
dp0p0s20f1       -                                 A/D
dp0p0s20f2       -                                 A/D
dp0p0s20f3       -                                 A/D
dp0vhost0        -                                 u/u
dp0vhost1        -                                 u/u
```

Using control structures

You can use control structures, such as conditionals and loops, by using the normal bash syntax. The vcli shell simply provides some of the required wrappers to enable the CLI commands to be scriptable. For more information about bash scripting, refer to http://wiki.bash-hackers.org/doku.php.

VCLI scripting examples

The following sample script, create-bridge.vcli, creates a bridge interface on a router.

```
#!/bin/vcli -f
```

```
configure
trap "{ end_configure; }" EXIT HUP
set interfaces bridge br1 description "bridge to nowhere"
set interfaces bridge br1 address 1.1.1.1/32
if ! validate; then
    exit 1
fi
if ! commit; then
    exit 1
fi
run show interfaces bridge br1
run show bridge br1
```

The following example shows how to run the create-bridge.vcli script on a router and shows the output of the script.

```
vyatta@vyatta# ./create-bridge.vcli
br1@NONE: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
 LOWERLAYERDOWN group default
    link/ether e2:4d:d8:13:c9:38 brd ff:ff:ff:ff:ff:ff
    inet 1.1.1.1/32 scope global br1
       valid_lft forever preferred_lft forever
    inet6 fe80::e04d:d8ff:fe13:c938/64 scope link
       valid_lft forever preferred_lft forever
    Description: bridge to nowhere
    RX:  bytes    packets     errors    ignored    overrun      mcast
             0          0          0          0          0          0
    TX:  bytes    packets     errors    dropped    carrier collisions
           188          2          0          0          0          0
bridge name    bridge id              STP enabled    interfaces
br1            0000.000000000000      no
[edit]
```

The following sample script (show-dataplane-IP-addresses.vcli) shows the IP addresses of the configured data plane interfaces.

```
#!/bin/vcli -f
configure
echo
echo "List of configured data plane interfaces and their corresponding $
echo "-------------------------------------------------------------------$
for i in $(list interfaces dataplane); do
  echo -n "$i:"
  addr=($(list interfaces dataplane $i address))
  echo ${addr[@]}
  if [ -z ${addr[@]} ]; then
    show interfaces dataplane $i address
  fi
done
echo
end_configure
```

The following example shows how to run the show-dataplane-IP-addresses.vcli script on a router and shows the output of the script.

```
vyatta@vyatta:~$ vcli -f show-dataplane-IP-addresses.vcli

List of configured data plane interfaces and their corresponding IP
 addresses:
-------------------------------------------------------------------------
---
dp0s160:10.18.170.205/24
```