NUAAL Documentation

Release 0.1.4-beta

Miroslav Hudec

May 24, 2018

CONTENTS

This project aims to bring multiple ways of communicating with networking gear under one unified API. This should allow network engineers to gather information about the network configuration in a fast and simple manner. In the early stage, this project focuses mainly on retrieving the information from the devices, not so much on performing configuration changes.

Installation

You can install NUAAL either from Python package index PyPI or use the source code files available at GitHub. To install NUAAL by pip, simply type `pip install nuaal` in your terminal.

Usage

For specific usage examples please check out appropriate section of this documentation. A good starting point describing how to get structured data directly from routers and switches is *Cisco_IOS_Cli*.

CHAPTER

ONE

CONNECTIONS

This section describes various ways to communicate with network devices in order to pull configuration information.

1.1 CLI Connection

CLI Connections use widely used protocols, SSH and Telnet in order to communicate with network device. However, the CLI is made to be readable by humans, not so much by machines. In order to overcome this problem, we need to use some form of parsing the text outputs into more machine-friendly format. Nuaal contains parsing library which is able to produce structured JSON format from these text outputs. Each CLI Connection class is responsible for two key procedures:

1. Connecting to device, sending appropriate command and retrieve corresponding text output
2. Parse the text output by using *Parsers* library and return data in JSON format.

Each device type is represented by specific object. For example, to connect to Cisco IOS (IOS XE) device, you need to use the *Cisco_IOS_Cli* object. To make finding the correct object easier, you can use the *GetCliHandler* function.

GetCliHandler(*device_type=None*, *ip=None*, *username=None*, *password=None*, *parser=None*, *secret=None*, *enable=False*, *store_outputs=False*, *DEBUG=False*)

This function can be used for getting the correct connection object for specific device type.

Parameters

- **device_type** (*str*) – String representation of device type, such as *cisco_ios*
- **ip** – (str) IP address or FQDN of the device you're trying to connect to
- **username** – (str) Username used for login to device
- **password** – (str) Password used for login to device
- **parser** – (ParserModule) Instance of ParserModule class which will be used for parsing of text outputs. By default, new instance of ParserModule is created.
- **secret** – (str) Enable secret for accessing Privileged EXEC Mode
- **enable** – (bool) Whether or not enable Privileged EXEC Mode on device
- **store_outputs** – (bool) Whether or not store text outputs of sent commands
- **DEBUG** – (bool) Enable debugging logging

Returns Instance of connection object

1.1.1 CliBaseConnection

class CliBaseConnection(*ip=None*, *username=None*, *password=None*, *parser=None*, *secret=None*, *enable=False*, *store_outputs=False*, *DEBUG=False*)

Bases: object

This class represents the base object, from which other (vendor specific classes) inherit. This class is basically a wrapper class around Kirk Byers' excellent library, netmiko. Even though the netmiko library already provides pretty straightforward and easy way to access network devices, the CliBaseConnection tries to handle multiple events which can arise, such as:

- Device is unreachable
- Fallback to Telnet if SSH is not supported by device (and vice-versa)
- Handles errors in outputs

Apart from the 'send command, receive output' this class also performs the parsing and storing outputs.

Parameters

- **ip** – (str) IP address or FQDN of the device you're trying to connect to
- **username** – (str) Username used for login to device
- **password** – (str) Password used for login to device
- **parser** – (ParserModule) Instance of ParserModule class which will be used for parsing of text outputs. By default, new instance of ParserModule is created.
- **secret** – (str) Enable secret for accessing Privileged EXEC Mode
- **enable** – (bool) Whether or not enable Privileged EXEC Mode on device
- **store_outputs** – (bool) Whether or not store text outputs of sent commands
- **DEBUG** – (bool) Enable debugging logging

_check_enable_level(*device*)

This function is called at the end of `self._connect()` to ensure that the connection is actually alive and that the proper privilege level is set.

Parameters **device** – (`Netmiko.ConnectHandler`) Instance of `netmiko.ConnectHandler`. If the connection is working, this will be set as `self.device`

Returns `None`

_command_handler(*action*, *filter=None*)

This function tries to send multiple 'types' of given command and waits for correct output. This should solve the problem with different command syntax, such as 'show mac address-table' vs 'show mac-address-table' on different versions of Cisco IOS. When correct output is returned, it is then parsed and the result is returned.

Parameters **commands** (*list*) – List of command string to try, such as ['show mac-address-table', 'show mac address-table']

Returns JSON representation of command output

_connect()

This function handles connection to device, if primary method fails, it will try to connect using secondary method.

Returns `None`

_connect_ssh()

This function tries to establish connection with device via SSH

Returns (`netmiko.ConnectHandler`) device

_connect_telnet()

This function tries to establish connection with device via Telnet

Returns (`netmiko.ConnectHandler`) device

_get_provider()

Creates provider dictionary for Netmiko connection

Returns `None`

_send_command(*command*, *expect_string=None*)

Parameters **command** (*str*) – Command to send to device

Returns Plaintext output of command from device

_send_commands(*commands*)
Sends multiple commands to device.

Parameters **commands** (*list*) – List of commands to run

Returns Dictionary with key=command, value=output_of_the_command

check_connection()
This function can be used to check state of the connection. Returns *True* if the connection is active and *False* if it isn't.

Returns Bool value representing the connection state.

config_mode()

disconnect()
This function handles graceful disconnect from the device.

Returns None

get_arp()
Returns content of device ARP table in JSON format. In Cisco terms, this represents the command *show ip arp*.

Returns List of dictionaries.

get_interfaces()
This function returns JSON representation of all physical and virtual interfaces of the device, containing all available info about each interface. In Cisco terms, this represents usage of command *show interfaces*.

Returns List of dictionaries.

get_inventory()
This function return JSON representation of all installed modules and HW parts of the device. In Cisco terms, this represents the command *show inventory*.

Returns List of dictionaries.

get_license()
This function return JSON representation of licenses activated or installed on the device. In Cisco terms, this represents the *show license* command.

Returns List of dictionaries.

get_mac_address_table()
Returns content of device MAC address table in JSON format. In Cisco terms, this represents the command *show mac address-table*.

Returns List of dictionaries.

get_portchannels()
This function returns JSON representation of all logical bind interfaces (etherchannels, portchannels). In Cisco terms, this represents the *show etherchannel summary* command.

Returns List of dictionaries.

get_version()
Returns JSON representation of basic device information, such as vendor, device platform, software version etc. In Cisco terms, this represents the command *show version*.

Returns List of dictionaries.

get_vlans()
This function returns JSON representation of all VLANs enabled on the device, together with list of assigned interfaces. In Cisco terms, this represents the *show vlan brief* command.

Returns List of dictionaries.

store_raw_output(*command*, *raw_output*, *ext='txt'*)
This function is used for storing the plaintext output of the commands called on the device in separate files. Used mainly for debugging and development purposes. This function is only called if the *store_outputs* parameter is set to *True*.

Parameters

- **command** (*str*) – Command string executed on the device.
- **raw_output** (*str*) – Plaintext output of the command.
- **ext** (*str*) – Extension of the file, ".txt" by default.

Returns None

1.1.2 Cisco_IOS_Cli

This class provides low-level connection to Cisco networking devices running Cisco IOS and IOS XE. In order for this connection to work, SSH or Telnet must be enabled on the target device. After creating connection object and connecting to the device using either Python's Context Manager or by calling _connect() method, you can start retrieving structured data by calling *get_* functions. To see other supported *get_* functions, please check out the documentation page of the parent object *CliBaseConnection*.

Example usage:

```
>>> from nuaal.connections.cli import Cisco_IOS_Cli
>>> # For further easier manipulation, create provider dictionary
>>> provider = {
    "username": "admin",     # Username for authentication
    "password": "cisco",     # Password for authentication
    "enable": True,          # Whether or not enable Privileged EXEC Mode
    "secret": "cisco",       # Enable secret for entering Privileged EXEC Mode
    "store_outputs": True,   # Enables saving Plaintext output of commands
    "method": "ssh"          # Optional, defaults to "ssh"
}
>>>
>>> # Create variable for storing data
>>> data = None
>>> # Establish connection using Context Manager
>>> with Cisco_IOS_Cli(ip="192.168.1.1", **provider) as ios_device:
>>>     # Run selected commands
>>>     ios_device.get_interfaces()
>>>     ios_device.get_vlans()
>>>     ios_device.get_trunks()
>>>     ios_device.get_version()
>>>     ios_device.get_inventory()
>>>     # Store data to variable
>>>     data = ios_device.data
>>> # Do other fancy stuff with data
>>> print(data)
```

Defined get_* functions return output in JSON format (except for get_config()). You can also send your own commands, such as:

```
>>> with Cisco_IOS_Cli(ip="192.168.1.1", **provider) as ios_device:
>>>     ios_device._send_command(command="show version")
```

And you will receive Plaintext output of selected command.

class Cisco_IOS_Cli(*ip=None*, *username=None*, *password=None*, *parser=None*, *secret=None*, *method='ssh'*, *enable=False*, *store_outputs=False*, *DEBUG=False*)

Bases: `nuaal.connections.cli.CliBase.CliBaseConnection`

Object for interaction with network devices running Cisco IOS (or IOS XE) software via CLI interface.

Parameters

- **ip** – (str) IP address or FQDN of the device you're trying to connect to
- **username** – (str) Username used for login to device
- **password** – (str) Password used for login to device
- **parser** – (ParserModule) Instance of ParserModule class which will be used for parsing of text outputs. By default, new instance of ParserModule is created.
- **secret** – (str) Enable secret for accessing Privileged EXEC Mode
- **method** – (str) Primary method of connection, 'ssh' or 'telnet'. (Default is 'ssh')
- **enable** – (bool) Whether or not enable Privileged EXEC Mode on device
- **store_outputs** – (bool) Whether or not store text outputs of sent commands
- **DEBUG** – (bool) Enable debugging logging

get_config()

Function for retrieving current configuration of the device.

Return str Device configuration

get_neighbors(*output_filter=None*, *strip_domain=False*)

Function to get neighbors of the device with possibility to filter neighbors :param output_filter: (Filter) Instance of Filter class, used to filter neighbors, such as only "Switch" or "Router" :param strip_domain: (bool) Whether or not to strip domain names and leave only device hostname :return: List of dictionaries

get_trunks(*expand_vlan_groups=False*)

Custom parsing function for output of "show interfaces trunk" :param expand_vlan_groups: (bool) Whether or not to expand VLAN ranges, for example '100-102' -> [100, 101, 102] :return: List of dictionaries

1.1.3 CliMultiRunner

class CliMultiRunner(*provider*, *ips*, *actions=None*, *workers=4*, *DEBUG=False*)

Bases: `object`

This class allows running set of CLI commands on multiple devices in parallel, using Worker threads

Parameters

- **provider** (*dict*) – Dictionary with necessary info for creating connection
- **ips** (*list*) – List of IP addresses of the device
- **actions** (*list*) – List of actions to be run in each connection
- **workers** (*int*) – Number of worker threads to spawn
- **DEBUG** (*bool*) – Enables/disables debugging output

fill_queue()

This function builds *device-specific* providers for CliConnection and adds them to Queue object.

Returns `None`

run()

Main entry function, puts all pieces together. Firs populates Queue with IP addreses and connection information, spawns threads and starts them. Blocks until `self.queue` is empty.

Returns

thread_factory()
Function for spawning worker threads based on number of workers in `self.workers`

Returns

worker()
Worker function to handle individual connections. Based on provider object from Queue establishes connection to device and runs defined set of commands. Received data are stored in `self.data` as a list of dictionaries, each dictionary representing one device.

Returns `None`

1.2 REST Connections

REST API Connections are used for communication with various network controllers, vedor-specific APIs and network devices, which support this method of communication.

1.2.1 RestBase Connection

class RestBase(*url*, *username=None*, *password=None*, *api_base_path=None*, *verify_ssl=False*, *DEBUG=False*, *con_type=None*)
Bases: `object`

Parent class for all REST-like connections

Parameters

- **url** – URL or IP address of the target machine
- **username** – Username for authentication
- **password** – Password for authentication
- **api_base_path** – Base path for the API resource, such as "/api/v1"
- **verify_ssl** – Enable SSL certificate verification. For self-signed certificates, set this to False
- **DEBUG** – Enable debugging output
- **con_type** – String representation of connection type, set by child classes

_authorize()
Connection-specific function for handling authorization. Overridden by child classes

Returns `None`

_delete(*path*, *params*)
Wrapper function for DELETE method of the requests library

Parameters

- **path** – Path of the API resource used in URL
- **params** – Parameters for the request

Returns Instance of `requests` response object

_get(*path*, *params=None*)
Wrapper function for GET method of the requests library

Parameters

- **path** – Path of the API resource used in URL
- **params** – Parameters for the request

Returns Instance of `requests` response object

_initialize()
Function for credentials loading and session preparation. Might be overwritten in child classes

Returns `None`

_post(*path*, *data=None*, *files=None*, *params=None*)
Wrapper function for POST method of the requests library

Parameters

- **path** – Path of the API resource used in URL
- **data** – Data payload for POST request. JSON string
- **files** (*dict*) – Dictionary with files to upload
- **params** – Parameters for the request

Returns Instance of `requests` response object

_put(*path*, *data=None*, *files=None*, *params=None*)
Wrapper function for PUT method of the requests library

Parameters

- **path** – Path of the API resource used in URL
- **data** – Data payload for POST request. JSON string
- **params** – Parameters for the request

Returns Instance of `requests` response object

_response_handler(*response*)
Function for handling request's response objects. Main purpose of this action is to handle HTTP return codes. In case of JSON formatted reply, returns this data as dictionary. In case of errors, either string representation of data is returned, or None. Status code is returned alongside the content to allow further processing if needed.

Parameters **response** – Instance of request's response object.

Returns content, status code

1.2.2 APIC-EM Base Connection

class ApicEmBase(*url*, *username=None*, *password=None*, *api_base_path='/api/v1'*, *verify_ssl=True*, *DEBUG=False*)
Bases: `nuaal.connections.api.RestBase.RestBase`

Object providing access to Cisco's APIC-EM REST-API.

Parameters

- **url** – URL of the APIC-EM, such as 'https://sandboxapicem.cisco.com'
- **username** – Username for authentication
- **password** – Password for authentication
- **api_base_path** – Base path for version-1 API
- **verify_ssl** – Enable/disable verification of SSL Certificate
- **DEBUG** – Enable/disable debugging output

_authorize()
Function for obtaining authorization "Service Ticket" from the APIC-EM. This function is called when HTTP Status Code 401 (Unauthorized) is returned.

Returns `None`

_load_credentials()
Function for loading credentials information stored in data/apicem_credentials.json file.

Returns None

_store_credentials()
Function for storing credentials information in data/apicem_credentials.json file. This function is called after successfully obtaining Service Ticket

Returns None

check_file(*fileId=None*, *namespace=None*, *file_name=None*)
Function for checking whether given file (based on it's name or ID) exists on the APIC-EM. Can be used for checking configuration or template files.

Parameters

- **fileId** (*str*) – ID string of the file in APIC-EM's database
- **namespace** (*str*) – APIC-EM's namespace, under which the file should be located
- **file_name** (*str*) – Name of the file

Return str FileID

delete(*path*, *params=None*)

Parameters

- **path** (*str*) – Path of API resource
- **params** (*dict*) – Dictionary of parameters for request

Returns Dictionary representation of response

delete_file(*fileId=None*, *namespace=None*, *filename=None*)
Function for deleting file from APIC-EM's database based on it's ID

Parameters

- **fileId** (*str*) – ID string of the file in APIC-EM's database
- **namespace** (*str*) – APIC-EM's namespace, under which the file should be located
- **filename** (*str*) – Name of the file

Return bool

get(*path*, *params=None*)
Function providing HTTP GET method for retrieving data from APIC-EM API.

Parameters

- **path** – Path of API resource, such as "/network-device". Each path must begin with "/" forward-slash character
- **params** – Dictionary of parameters for request, such as {"deviceId": "<ID of device in APIC-EM database>"}

Returns Dictionary representation of response content under "response" key, eg. <request_response_object>.json()["response"] or None

post(*path*, *data=None*, *files=None*, *params=None*)
Function providing HTTP POST method for sending data to APIC-EM API.

Parameters

- **path** (*str*) – Path of API resource, such as "/ticket". Each path must begin with "/" forward-slash character
- **data** (*json*) – JSON string data payload
- **files** (*dict*) – Dictionary with files to upload

- **params** – Dictionary of parameters for request, such as {"deviceId": "<ID of device in APIC-EM database>"}

Returns Dictionary representation of response content under "response" key, eg. <request_response_object>.json()["response"] or `None`

put (*path*, *data=None*, *files=None*, *params=None*)
Function providing HTTP PUT method for sending data to APIC-EM API.

Parameters

- **path** (*str*) – Path of API resource
- **data** (*json*) – JSON string data payload
- **files** (*dict*) – Dictionary with files to upload
- **params** (*dict*) – Dictionary of parameters for request, such as {"deviceId": "<ID of device in APIC-EM database>"}

Returns Dictionary representation of response content under "response" key, eg. <request_response_object>.json()["response"] or `None`

task_handler (*taskId*, *timeout=2*, *max_retries=2*)
Function for handling Task results, based on taskId. This function periodically queries APIC-EM for the results of given task until task is either completed or timeout is reached. Maximum time this function waits for result is (timeout * max_retries) seconds.

Parameters

- **taskId** (*str*) – ID of the task, returned by the APIC-EM after starting task.
- **timeout** (*int*) – Number of seconds to wait after each unsuccessful request
- **max_retries** (*int*) – Maximum number of requests.

Return str ID of the result

upload_file (*namespace*, *file_path*)
Function for uploading file to APIC-EM's database.

Parameters

- **namespace** (*str*) – APIC-EM's namespace, under which the file should be uploaded
- **file_path** (*str*) – Path to the file on local machine

Return str FileID of the uploaded file

CHAPTER

TWO

PARSERS

This section describes classes for parsing CLI outputs of network devices to structured objects - dictionaries.

2.1 CLI Parsers

2.1.1 ParserModule

`ParserModule(object)`

This class represents the base object, from which other (vendor specific classes) inherit. This class provides basic set of functions which handle parsing of raw text outputs based on Python's `re` module and *regex* strings provided by *Patterns* class.

The core idea behind `ParserModule` is that the complexity of various command outputs differ. Some outputs, such as *'show mac address-table'* are simple enough to be handled by single *regex* pattern. On the other hand, some outputs, such as *'show interfaces'* are way more complex and would require a very long and complex *regex patterns* to match. Also, every little change in the output would cause this pattern not to match. To avoid the usage of overly complex regex patterns, the parsing can be divided into two steps, or *levels*. In the first *level*, the text output is 'pre-processed', for example the output of *show interfaces* is splitted to list of strings, where each string represents a single interface. In the second *level*, each of these strings can be processed using multiple patterns, resulting in final *dictionary* representing the interface. For some outputs, these two levels can "overlap", meaning that the first level can already return a *dictionary*, but some of the keys need to be 'post-processed' by the second level. A great example of such command is *'show vlan brief'*, where the first level returns a list of dictionaries with keys *["vlan_id", "name", "ports"]*. The value of ""ports"' (which is a string of all ports assigned to this VLAN) can be further processed by second *level*, which converts this string to a list of individual ports.

Base Functions

- `match_single_pattern(self, text, pattern)` - This function tries to match a single *regex* `pattern` on given `text` *string*. This function operates in two 'modes': with or without *named groups* in regex pattern. If `pattern` contains at least one *named group* (for example `r"^(?P<name_of_the_group>.*)"`), the function will return *list* of *dictionaries*, where each *dictionary* has *"name_of_the_group"* as key and whatever the `.*` matched as value. If *pattern* does not contain any *named group*, *list* of *strings* is returned, each string being one match of the pattern (basically `re.findall(pattern=pattern, string=text)`).
- `update_entry(self, orig_entry, new_entry)` - This function simply updates *dict* `orig_entry` with *dict* `new_entry`. The changes are made only for keys, that are not in the `orig_entry` or those, which value is `None`. The updated *dictionary* is returned.
- `match_multi_pattern(self, text, patterns)` - This function uses multiple *regex* `patterns` to match against given `text`. At the beginning, a new *dictionary* is created with *named groups* of ALL the patterns as keys and values `None`. Each time one of the patterns matches, the resulting *dictionary* is updated.
- `_level_zero(self, text, patterns)` - This function tries to match a pattern from *patterns* until a match is found. Then based on the `self.match_single_pattern(**kwargs)` returns either list

of strings, or list of dictionaries. This function is used either for matching *'simple'* outputs or for pre-processing (and post-processing) of more complex outputs.

- `_level_one(self, text, command)` - Given the command variable (which represents the command used to get output), it determines the *level* of the command and fetches corresponding *patterns* from `self.patterns["level0"][command]` (an instance of `Patterns` class). If the *level* is 0, it simply returns the output of `self._level_zero(text, patterns)`. If the *level* is 1, it continues to process individual entries returned by `_level_zero` based on patterns from `self.patterns["level1"][command]`. Return a list of dictionaries.
- `autoparse(self, text, command)` - The main entry point for parsing, given just the `text` output and `command` it determines the proper way to parse the output and returns result.

class ParserModule(*device_type*, *DEBUG=False*)

Bases: `object`

This class provides necessary functions for parsing plaintext output of network devices. Uses patterns from `PatternsLib` for specified device type. The outputs are usually lists of dictionaries, which contain keys based on name groups of used regex patterns.

Parameters

- **device_type** (*str*) – String representation of device type, such as *cisco_ios*
- **DEBUG** (*bool*) – Enables/disables debugging output

_level_one(*text*, *command*)

This function handles parsing of more complex plaintext outputs. First, the output of `_level_zero()` function is retrieved and then further parsed.

Parameters

- **text** (*str*) – Plaintex output of given `command`, which will be parsed
- **command** (*str*) – Command string used to generate the output

Returns List of dictionaries

_level_zero(*text*, *patterns*)

This function handles parsing of less complex plaintext outputs, which can be parsed in one step.

Parameters

- **text** (*str*) – Plaintex output which will be parsed
- **patterns** – List of compiled regex patterns, which are used to parse the `text`

Returns List dics (if `patterns` contain named groups) or list of strings (if they don't)

autoparse(*text*, *command*)

The main entry point for parsing, given just the `text` output and `command` it determines the proper way to parse the output and returns result.

Parameters

- **text** (*str*) – Text output to be processed
- **command** (*str*) – Command used to generate `text` output. Based on this parameter, correct regex patterns are selected.

Returns List of found entities, usually list of dictionaries

command_mapping(*command*)

This function determines the max_level of command - based on level of complexity of the output, the parsing is processed in 1 or 2 steps. Used for *autoparse*

Parameters **command** (*str*) – Command used to generate the output, eg. *show vlan brief*

Returns (str) Highest level of specified command (top PatternsLib Key)

match_multi_pattern(*text*, *patterns*)
This functions tries to match multiple regex `patterns` against given `text` .

Parameters

- **text** (*str*) – Text output to be processed
- **patterns** (*lst*) – List of `re` compiled patterns for parsing.

Returns Dictionary with names of all groups from *all* `patterns` as keys, with matching strings as values.

match_single_pattern(*text*, *pattern*)
This function tries to match given regex `pattern` against given `text`. If `pattern` contains named groups, list of dictionaries with these groups as keys is returned. If `pattern` does not contain any named groups, list of matching strings is returned.

Parameters

- **text** (*str*) –
- **pattern** – `re` compiled regex pattern

Returns List of matches, either dictionaries or strings

update_entry(*orig_entry*, *new_entry*)
This function simply updates dictionary `orig_entry` with keys and values from `new_entry`. Only `None` values in `orig_entry` are updated. Keys with value `None` from `new_entry` are also used in order to ensure coherent output.

Parameters

- **orig_entry** (*dict*) – Original dictionary to be updated based on entries in `new_entry`
- **new_entry** (*dict*) – New dictionary containing new data which should be added to `orig_entry`

Returns Updated dictionary

class CiscoIOSParser(*DEBUG=False*)
Bases: `nuaal.Parsers.Parser.ParserModule`

Child class of *ParserModule* designed for *cisco_ios* device type.

trunk_parser(*text*)
Function specifically designed for parsing output of *show interfaces trunk* command.

Parameters **text** (*str*) – Plaintext output of *show interfaces trunk* command.

Returns List of dictionaries representing trunk interfaces.

CHAPTER

THREE

MODELS

Models are high-level representation of network entities, such as devices, interfaces, network topologies etc. The main purpose of these *models* is to provide single unified interface producing outputs of specified structure, no matter what kind of connection is used on the lower level. This helps overcome issues with different dictionary structures, names of methods and more. Because each type of connection (CLI, REST API, Network Controller) provides structured, but always slightly different format of data, there exists a model for each type of connection. This model then converts *connection specific* data structures to those defined in BaseModel classes.

3.1 BaseModels

class BaseModel(*name=None*, *DEBUG=False*)

Bases: `object`

This is a parent object that all high-level API models inherit from.

Parameters

- **name** –
- **DEBUG** –

class DeviceBaseModel(*name='DeviceBaseModel'*, *DEBUG=False*)

Bases: *nuaal.Models.BaseModels.BaseModel*

This is a parent object for device models.

Parameters

- **name** –
- **DEBUG** –

get_interfaces()

Returns

get_neighbors()

Returns

get_vlans()

Returns

3.2 Cisco_IOS_Model

class CiscoIOSModel(*cli_connection=None*, *DEBUG=False*)

Bases: *nuaal.Models.BaseModels.DeviceBaseModel*

High-level abstraction model for Cisco IOS devices.

Parameters

- **cli_connection** – Instance of `Cisco_IOS_Cli` connection object.
- **DEBUG** (*bool*) – Enables/disables debugging output.

_interface_update()
Internal function for retrieving needed data to build complete interface representation.

Returns None

_map_interface(*interface*)
Internal function for changing the format of interface dictionary representation.

Parameters **interface** (*dict*) – Dictionary representation of interface in a format provided by connection object's `get_interfaces()` function.

Returns Dictionary representation interface in common format.

get_interfaces()
Retrieves list of all interfaces in common format.

Returns List of dictionaries.

get_inventory()
Retrieves list of all installed HW parts on device in common format.

Returns List of dictionaries.

get_l2_interfaces()
Retrieves list of L2 (switched) interfaces in common format.

Returns List of dictionaries.

get_l3_interfaces()
Retrieves list of L3 (routed) interfaces in common format.

Returns List of dictionaries.

get_vlans()
Retrieves list of all configured VLANs on device in common format.

Returns List of dictionaries.

3.3 ApicEmModels

class ApicEmDeviceModel(*apic=None*, *object_id=None*, *filter=None*, *DEBUG=False*)
Bases: *nuaal.Models.BaseModels.DeviceBaseModel*

Parameters

- **apic** –
- **object_id** –
- **filter** –
- **DEBUG** –

_initialize()

Returns

get_interfaces()

Returns

get_inventory()

Returns

get_vlans()

Returns

Lets look at the following example: In the first phase, we create two different connections, one being `Cisco_IOS_Cli` and the second being `ApicEmBase`:

```
>>> import json
>>> from nuaal.connections.cli import Cisco_IOS_Cli
>>> from nuaal.connections.api import ApicEmBase

>>> device1 = Cisco_IOS_Cli(**provider)
>>> device2 = ApicEmBase(url="https://sandboxapicem.cisco.com")
```

Both of these 'devices' have completely different functions and data structures. In order to get interfaces of the *device1*, one would simply call:

```
>>> interfaces_1 = device1.get_interfaces()
>>> # Print just the first interface for brevity
>>> for interface in interfaces_1[:1]:
>>> print(json.dumps(interface, indent=2))
{
 "name": "FastEthernet0",
 "status": "up",
 "lineProtocol": "up",
 "hardware": "Fast Ethernet",
 "mac": "1cdf.0f45.52e0",
 "bia": "1cdf.0f45.52e0",
 "description": "MGMT",
 "ipv4Address": null,
 "ipv4Mask": null,
 "loadInterval": "5 minute",
 "inputRate": 1000,
 "inputPacketsInterval": 1,
 "outputRate": 1000,
 "outputPacketsInterval": 1,
 "duplex": "Full",
 "speed": null,
 "linkType": null,
 "mediaType": null,
 "sped": "100Mb/s",
 "mtu": 1500,
 "bandwidth": 100000,
 # .
 # <Output ommited>
 # .
 "outputBufferSwappedOut": 0
}
>>>
```

But for the *device2* (which isn't actually single device, but a controller with many devices), we would run:

```
>>> # Presume already knowing the id of device
>>> interfaces_2 = device2.get(path="/interface/network-device/<deviceId>")
>>> for interface in interfaces[:1]:
>>> print(json.dumps(interface, indent=2))
{
   "className": "SwitchPort",
   "description": "",
   "interfaceType": "Physical",
   "speed": "1000000",
   "adminStatus": "UP",
   "macAddress": "70:81:05:42:1e:b3",
```

(continues on next page)

(continued from previous page)


```
    "ifIndex": "43",
    "status": "down",
    "voiceVlan": null,
    "portMode": "dynamic_auto",
    "portType": "Ethernet Port",
    "lastUpdated": "2018-05-02 14:53:01.67",
    "portName": "GigabitEthernet5/36",
    "ipv4Address": null,
    "ipv4Mask": null,
    "isisSupport": "false",
    "mappedPhysicalInterfaceId": null,
    "mappedPhysicalInterfaceName": null,
    "mediaType": "10/100/1000-TX",
    "nativeVlanId": "1",
    "ospfSupport": "false",
    "serialNo": "FOX1524GV2Z",
    "duplex": "AutoNegotiate",
    "series": "Cisco Catalyst 4500 Series Switches",
    "pid": "WS-C4507R+E",
    "vlanId": "1",
    "deviceId": "c8ed3e49-5eeb-4dee-b120-edeb179c8394",
    "instanceUuid": "0054cc51-ea16-471c-a634-5788220ff3f3",
    "id": "0054cc51-ea16-471c-a634-5788220ff3f3"
}
>>>
```


Apparently, both the provided information and data structure is different. To combine these data to single uniform database. This is where *models* come in. We will continue based on previous example, right after defining connections: `device1` and `device2`. From there on, we can create models:

```
>>> from nuaal.Models import Cisco_IOS_Model, ApicEmDeviceModel
>>> model1 = Cisco_IOS_Model(cli_connection=device1)
>>> model2 = ApicEmDeviceModel(apic=device2)
```

CHAPTER

FOUR

WRITERS

This section describes classes for writing data in various formats, such as CSV or Excel files.

4.1 Writer

class Writer(*type*, *DEBUG=False*)

Bases: `object`

This object handles writing structured data in JSON format in tabular formats, such as CSV or Microsoft Excel (.xlsx).

Parameters

- **type** (*str*) – String representation of writer type, used in logging messages.
- **DEBUG** (*bool*) – Enables/disables debugging output.

_get_headers(*data*)

This function returns sorted list of column headers (dictionary keys). :param list|dict data: List or dict of dictionaries containing common keys. :return: List of headers.

combine_data(*data*)

Function for combining data from multiple sections. Each section represents data gathered by some of the *get_* functions. Each returned dataset includes common headers identifying device, from which the data originates.

Parameters **data** (*list*) – List of dictionaries, content of single section

Returns (list) section_headers, (list) section_content

combine_device_data(*data*)

json_to_lists(*data*)

This function transfers list of dictionaries into two lists, one containing the column headers (keys of dictionary) and the other containing individual list of values (representing rows).

Parameters **data** (*list*) – List of dictionaries with common structure

Returns Dict with "headers" list and "list_data" list containing rows.

4.2 ExcelWriter

class ExcelWriter

Bases: `nuaal.Writers.Writer.Writer`

This class provides functions for writing the retrieved data in Microsoft Excel format, *.xlsx*.

create_workbook(*path*, *filename*)

Function for creating Excel Workbook based on path and file name.

Parameters

- **path** (*str*) – System path to Excel file
- **filename** (*str*) – Name of the Excel file.

Returns Instance of `xlsxwriter` workbook object.

write_data(*workbook*, *data*)
Function for writing entire content of device data, divided into sections based on *get_* command used. :param workbook: Reference of the `xlsxwriter` workbook object. :param dict data: Content of device.data variable, which holds all retrieved info. :return: `None`

write_json(*workbook*, *data*, *worksheetname=None*, *headers=None*)
Function for writing JSON-like data to worksheet.

Parameters

- **workbook** – Reference of the `xlsxwriter` workbook object.
- **data** –
- **worksheetname** (*str*) – Name of the worksheet
- **headers** (*list*) – List of column headers.

Returns `None`

write_list(*workbook*, *data*, *worksheetname=None*, *headers=None*)
Function for creating worksheet inside given workbook and writing provided data.

Parameters

- **workbook** – Reference of the `xlsxwriter` workbook object.
- **data** (*list*) – List of lists, where each list represents one row in the worksheet.
- **worksheetname** (*str*) – Name of the worksheet.
- **headers** (*list*) – List of column headers.

Returns `None`

CHAPTER

FIVE

DISCOVERY

This module provides classes for network discovery. Discovery can be performed by two ways:

- Using one device as seed for discovery based on discovery protocols
- Using list of IP addresses of all devices in the network

Note: Both mentioned ways rely on neighbor discovery protocols, which needs to be enabled in the network, such as CDP or LLDP. Without information provided by these protocols, it is impossible to reconstruct the network topology.

5.1 IP_Discovery

This class performs discovery of network devices based on known IP addresses. Can be used in scenarios where you want only specific devices to be discovered or when discovery protocols are not enabled in given network.

class IP_Discovery(*provider*, *DEBUG=False*)

Bases: `object`

This function performs discovery of the network devices based on their IP addresses.

Parameters

- **provider** (*dict*) – Provider dictionary containing information for creating connection object, such as credentials
- **DEBUG** (*bool*) – Enables debugging output

run(*ips*)

Main entry function. Starts the discovery process based on given IP address of the seed device.

Parameters **ips** (*list*) – List of IP addresses for discovery

Returns `None`

5.2 Neighbor_Discovery

This class performs discovery of network devices based on seed device's IP address. *IP_Discovery*

Note: In order for this discovery method to work, all network devices must have CDP (or LLDP) discovery protocol enabled on all links which connect to other devices. Also make sure that the advertised IP address is the IP address intended for device management and is reachable from the node you're running NUAAL on.

class Neighbor_Discovery(*provider*, *DEBUG=False*)
Bases: object

This class provides a simple way to perform network discovery based on CDP neighbors of device. Given IP address of initial device (or 'seed device') it tries to crawl trough ne network and discover all supported devices. CDP must be enabled on devices.

Parameters

- **provider** (*dict*) – Provider dictionary containing information for creating connection object, such as credentials
- **DEBUG** (*bool*) – Enables/disables debugging output

_gen_device_id(*ip*, *hostname=None*)
Function to generate pseudo-unique identification of device. By default, it returns device hostname, if hostname is one of the default hostnames, it uses IP address to distinguish devices with same hostname.

Parameters

- **ip** – String - IP address of the device
- **hostname** – String (optional) - Hostname of the device

Returns String - such as "C2960X_AB01" or "Router(192.168.1.1)"

get_neighbors(*ip*, *hostname=None*)
This function performs the discovery of the neighbors for given device. Results are stored in self.to_process. After each round, the results are processed by self.process_neigbors()

Parameters

- **ip** – String - IP Address of the device
- **hostname** – String (optinal) - Hostname of the device, if not given, will be set after connecting to device

Returns None

process_neighbors()
This functions decides what to do with neighbors of device, such as which were already discovered or visited. It runs after every discovery round to combine data retrieved by individual worker threads.

Returns None

run(*ip*)
This function starts the discovery process based on given IP address of the *seed* device.

Parameters **ip** (*str*) – IP address of the seed device

Returns None

worker()
This is a wrapper function that is run as thread.

Returns None

5.3 Topology

Topology objects are used to build JSON representation of network topologies based on information of device neighbors. These classes use the `get_neighbors()` function of low-level connection classes, such as *Cisco_IOS_Cli*.

class Topology(*DEBUG=False*)
Bases: object

next_ui()

5.4 CliTopology

class CliTopology(*DEBUG=False*)

Bases: `nuaal.Discovery.Topology.Topology`

_get_links(*device_id*, *neighbors*)

_reverse_link(*link*)

build_topology(*data*)

CHAPTER

SIX

UTILS

get_logger(*name, DEBUG=False, handle=['stderr']*)

This function provides common logging facility by creating instances of *loggers* from python standard `logging` library.

Parameters

- **name** (*str*) – Name of the logger
- **DEBUG** (*bool*) – Enables/disables debugging output
- **handle** (*list*) – Changing value of this parameter is not recommended.

Returns Instance of logger object

interface_split(*interface*)

Function for splitting names of interfaces to interface type and number. Example: *FastEthernet0/18* -> *FastEthernet*, *0/18* :param str interface: Interface name. :return: Interface type, interface number

vlan_range_shortener(*full_vlan_list*)

Function for shortening list of allowed VLANs on trunk interface. Example: *[1,2,3,4,5]* -> *["1-5"]*. Reverse function is `vlan_range_expander`.

Parameters **full_vlan_list** – List of integers representing VLAN numbers.

Returns List of strings.

vlan_range_expander(*all_vlans*)

Function for expanding list of allowed VLANs on trunk interface. Example: *1-4096* -> range(1, 4097). Can be used when trying to figure out whether certain VLAN is allowed or not. Reverse function is `vlan_range_shortener`.

Parameters **all_vlans** – Either list (*["1-10", "15", "20-30"]*) or string (*"1-10,15,20-30"*) of VLAN range.

Returns List of VLAN numbers (integers).

int_name_convert(*int_name*)

Function for converting long interface names to short and vice versa. Example: *"GigabitEthernet0/12"* -> *"Gi0/12"* and *"Gi0/12"* -> *"GigabitEthernet0/12"*

Parameters **int_name** (*str*) – Interface name

Returns Interface name

mac_addr_convert(*mac_address=''*)

Function for providing single format for MAC addresses.

Parameters **mac_address** (*str*) – MAC address to be converted.

Returns MAC address in format *XX:XX:XX:XX:XX:XX*.

update_dict(*orig_dict*, *update_dict*)

Function for updating dictionary values.

Parameters

- **orig_dict** (*dict*) – Dictionary to be updated.
- **update_dict** (*dict*) – Dictionary to update from.

Returns Updated dictionary.

check_path(*path*, *create_missing=True*)
Function for checking path availability, returns *path* if specified folder exists, `False` otherwise. Can also create missing folders. Used for handling absolute and relative paths.

Parameters

- **path** (*str*) – Path of the folder.
- **create_missing** (*bool*) – Whether or not to create missing folders.

Returns Bool

6.1 Filter

class Filter(*required={}*, *excluded={}*, *exact_match=True*, *DEBUG=False*)
Bases: `object`

Object for making filtering of required field easier.

This class finds elements in JSON-like format based on provided values.

Parameters

- **required** (*dict*) – Dictionary of required keys. Example: *{"key1": "value1"}* will return only those dictionaries, which key *"key1"* contains the value *"value1"*. You can also use list of required values, such as *{"key1": ["value1", "value2"]}* together with option `exact_match=False`.
- **excluded** (*dict*) – Dictionary of excluded keys. Example: *{"key1": "value1"}* will return only those dictionaries, which key *"key1"* does not contain the value *"value1"*. You can also use list of excluded values, such as *{"key1": ["value1", "value2"]}* together with option `exact_match=False`.
- **exact_match** (*bool*) – Specifies whether the filter value must match exactly, or partially.
- **DEBUG** – Enables/disables debugging output.

dict_cleanup(*data*)
Function for filtering dictionary structures (eg. dictionary where values are also dictionaries).

Parameters **data** (*dict*) – Dictionary containing dicts as values, which you want to filter.

Returns Filtered dictionary.

list_cleanup(*data*)
Function for filtering list structures (eg. list of dictionaries).

Parameters **data** (*list*) – List of dictionaries, which you want to filter.

Returns Filtered list of dictionaries.

universal_cleanup(*data=None*)
This function calls proper cleanup function base on data type.

Parameters **data** – Data to be filtered, either list of dictionaries or dictionary of dictionaries.

Returns Filtered data

6.2 OutputFilter

class OutputFilter(*data=None*, *required=[]*, *excluded=[]*)

Bases: `object`

This class helps to minimize dictionary structure by specifying only the desired keys.

Parameters

- **data** – Data to be filtered.
- **required** (*list*) – List of required keys. Returned entries will contain only these specified keys. Example: *{"key1": "value1", "key2": "value2"}* with `required` *["key1"]* will only return *{"key1": "value1"}*.
- **excluded** (*list*) – List of excluded keys. Returned entries will not contain these specified keys. Example: *{"key1": "value1", "key2": "value2"}* with `excluded` *["key1"]* will only return *{"key2": "value2"}*.

get()

After instantiating object, call this function to retrieve filtered data.

Returns Filtered data.

INDEX