unificontrol Documentation

Release 0.2.8

Nicko van Someren

Dec 10, 2020

Contents:

1 A high-level Python interface to the Unifi controller software **3**
- 1.1 Installation . . . 3
- 1.2 Documentation . . . 3
- 1.3 Usage . . . 4
- 1.4 Support for self-signed certificates . . . 4
- 1.5 Acknowledgments . . . 4

2 The unificontrol API **5**
- 2.1 Data types . . . 5
- 2.2 The UnifiClient class . . . 6
- 2.3 Constants . . . 15

3 SSL Security with self-signed certificates **17**

4 Extending the Unifi API **19**
- 4.1 Metaprogramming . . . 19
- 4.2 Implementing API calls . . . 19
- 4.3 JSON fix-up methods . . . 21

5 Indices and tables **23**

Python Module Index **25**

Index **27**

unificontrol is a rich and full-featured Python interface to the Ubiquiti Unifi software defined network controller. Goals of this package include:

- A clean interface that supports introspection and self-documentation.
- A complete implementation of the Unifi controller API. Currently it supports over 100 API calls to the Unifi controller.
- Proper handling of SSL connections to allow secure access to the controller even when the controller uses a self-signed certificate.
- A concise, readable internal representation of the Unifi API, so that new API calls can easily be added as new features are added to the controller.
- Python 3 only, since it's the way of the future.

This is a reference to *login()* to check what works.

CHAPTER 1

A high-level Python interface to the Unifi controller software

unificontrol is a rich and full-featured Python interface to the Ubiquiti Unifi software defined network controller. Goals of this package include:

- A clean interface that supports introspection and self-documentation.
- A complete implementation of the Unifi controller API. Currently it supports over 100 API calls to the Unifi controller.
- Proper handling of SSL connections to allow secure access to the controller even when the controller uses a [self-signed certificate](ssl_self_signed.md).
- A concise, readable internal representation of the Unifi API, so that new API calls can easily be added as new features are added to the controller.
- Python 3 only, since it's the way of the future.

1.1 Installation

To install the most recent release use:

```
pip install unificontrol
```

To install the latest version of the code from GitHub use:

```
pip install -e git+https://github.com/nickovs/unificontrol.git@master#egg=unificontrol
```

1.2 Documentation

The *unificontrol* code aims to be self-documenting as far as possible so if you are using it in an interactive environment the built in Python *help()* function will often tell you what you need.

There is also documentation that can be built using Sphynx in the *docs* directory and a built version of these docs is hosted on ReadTheDocs.

1.3 Usage

The simplest way to use this client is simply to create an instance with the necessary parameters and log in:

```
client = UnifiClient(host="unifi.localdomain",
    username=UNIFI_USER, password=UNIFI_PASSWORD, site=UNIFI_SITE)
```

The host name (and the host port, if you are using something other than the default 8443) must be specificed when you create the client. The username and password can be passed to the login method instead of the contstructor if you prefer. If you supply then username and password in the constructor then the client will automatically log in when needed and re-authenticate if your session expires.

Once you have created a client object you can simply make calls to the various API endpoints on the controler:

```
# Get a list of all the guest devices for the last day
guests = client.list_guests(within=24)

# Upgrade one of the access points
client.upgrade_device("11:22:33:44:55:66")
```

See the *API documentation* for full details.

1.4 Support for self-signed certificates

Since the Unifi controller uses a *self-signed certifcate* the default behaviour of the client is to fetch the SSL certificate from the server when you create the client instance and pin all future SSL connections to require the same certificate. This works OK but if you are building some tool that will talk to the controller and you have place to store configuration then a better solution is to store a copy of the correct certificate in a safe place and supply it to the constructor using the *cert* keyword argument. A server's certifcate can be fetched using the python ssl library:

```
import ssl
cert = ssl.get_server_certificate(("unifi.localdomain", 8443))
# Store the cert in a safe place
...
# Fetch the cert from a safe place
client = UnifiClient(host="unifi.localdomain",
    username=UNIFI_USER, password=UNIFI_PASSWORD, site=UNIFI_SITE,
    cert=cert)
```

If you have a proper certificate for the controller, issued by a known authority and with a subject name matching the host name used to access the server then you can switch off the certificate pinning by passing `cert=None`.

1.5 Acknowledgments

I would almost certainly never have written such a complete implementation of the API had it not been for the hard work done by the authors of the PHP Unifi API client created by Art of WiFi. While the code here was written from scratch, all of the necessary analysis and understanding of the undocumented API was taken from the PHP client. Without that open source project I would probably have stopped with less than a quarter of the API finished.

CHAPTER 2

The unificontrol API

Interaction with Unifi controllers is done by creating an instance of *unificontrol.UnifiClient*. The methods of this class represent calls to the various APIs exposed by the controller.

2.1 Data types

Most of the data types used in the API are fairly self-explanitory. There are however a few cases where some explaination is necessary.

2.1.1 ID values

In many of the API calls various entities such as networks, user groups, managed devices or other items are referred to by an ID value. In most cases these are 24 character unique hexadecimal strings which bear no relation to the visible names of these objects. In these cases you will need to use the various `list_...` methods to get lists of the available objects and use the `_id` attribute from the object you need.

2.1.2 Settings dictionaries

Many of the `set_site_...` calls take a `settings` dictionary. In these case the *list_settings* method can be used to find the current settings object and thus determine the keys expected in the settings dictionary.

Note: The settings dictionary should NOT contain an entry with the key `_id` as this will be automatically assigned. You should also remove the entry with the key `key` as this will be set to internally to refect the type of site setting to be set.

2.2 The UnifiClient class

class UnifiClient(*host='localhost'*, *port=8443*, *username='admin'*, *password=None*, *site='default'*, *cert='FETCH_CERT'*)

An abstract interface to the Unifi controller

Parameters

- **host** (*str*) – hostname of Unifi controller
- **port** (*int*) – port on which controller is to be accessed
- **username** (*str*) – user name for admin account
- **password** (*str*) – password for admin account
- **site** (*str*) – identifier of site to be managed
- **cert** (*str or bytes*) – Server SSL certificate to pin for secure access. Pass `None` to use regular certificate verification or the constant `FETCH_CERT` to use the current certificate of the server and pin that cert for future accesses.

host

Host name of contoller

Type str

port

Port for accessing controller

Type str

site

Identifier of site being managed

Type str

login(*username=None*, *password=None*)

Log in to Unifi controller

Parameters

- **username** (*str*) – *optional* user name for admin account
- **password** (*str*) – *optional* password for admin account

The username and password arguments are optional if they were provided when the client was created.

logout()

Log out from Unifi controller

authorize_guest(*mac*, *minutes*, *up=None*, *down=None*, *MBytes=None*, *ap_mac=None*)

Authorize a client device

Parameters

- **mac** (*str*) – MAC address of the guest client to be authorized
- **minutes** (*int*) – duration for which the client is authorised
- **up** (*int*) – *optional* upstream bandwidth limit in Kb/sec
- **down** (*int*) – *optional* downstream bandwidth limit in Kb/sec
- **MBytes** (*int*) – *optional* total data volume limit in megabytes
- **ap_mac** (*str*) – *optional* MAC address of the access point to which the client will attach

unauthorize_guest(*mac*)
Unauthorize a guest client device

Parameters **mac** (*str*) – MAC address of guest client to unauthorize

reconnect_client(*mac*)
Force reconnection of a client device

Parameters **mac** (*str*) – MAC address of guest client to reconnect

block_client(*mac*)
Block a client device

Parameters **mac** (*str*) – MAC address of guest client to block

unblock_client(*mac*)
Unblock a client device

Parameters **mac** (*str*) – MAC address of guest client to unblock

forget_client(*macs*)
Forget a client device

Parameters **mac** (*str*) – One or a litst of MAC addresses of guest clients to forget

---

Note: Requires version 5.9 of the controller or later.

---

create_client(*mac*, *usergroup_id*, *name=None*, *note=None*)
Creat a new user/client device

Parameters

- **mac** (*str*) – MAC address of new client
- **usergroup_id** (*str*) – _id value for the user group for the client
- **name** (*str*) – *optional* name for the new client
- **note** (*str*) – *optional* note to attach to the new client

set_client_note(*user_id*, *note*)
Add, modify or remove a note on a client device

Parameters

- **user_id** (*str*) – _id value of the user for which the note is set
- **note** (*str*) – Note to attach, or None to remove note

set_client_name(*user_id*, *name*)
Add, modify or remove a name of a client device

Parameters

- **user_id** (*str*) – _id value of the user for which the name is set
- **name** (*str*) – name to attach, or None to remove name

set_client_fixed_ip(*user_id*, *fixed_ip*, *network_id*)
Add, modify or remove a fixed ip of a client device

Parameters

- **user_id** (*str*) – _id value of the user for which the name is set
- **fixed_ip** (*str*) – IP to attach, or None to remove IP

- **network_id** (*str*) – network to attach

stat_5minutes_site(*start=None, end=None, attrs=['bytes', 'wan-tx_bytes', 'wan-rx_bytes', 'wlan_bytes', 'num_sta', 'lan-num_sta', 'wlan-num_sta', 'time']*)
Fetch site statistics with 5 minute granularity

Parameters

- **start** (*int*) – *optional* start of reporting period, as seconds in the Unix epoch. If not present defaults to 12 hours before the end time
- **end** (*int*) – *optional* end of reporting period, as seconds in the Unix epoch. If not present defaults to the current time.
- **attrs** (*list*) – *optional* list of statistics to return

Returns List of dictionaries of statistics

stat_hourly_site(*start=None, end=None, attrs=['bytes', 'wan-tx_bytes', 'wan-rx_bytes', 'wlan_bytes', 'num_sta', 'lan-num_sta', 'wlan-num_sta', 'time']*)
Fetch site statistics with 1 hour granularity

Parameters

- **start** (*int*) – *optional* start of reporting period, as seconds in the Unix epoch. If not present defaults to 7 days before the end time
- **end** (*int*) – *optional* end of reporting period, as seconds in the Unix epoch. If not present defaults to the current time.
- **attrs** (*list*) – *optional* list of statistics to return

Returns List of dictionaries of statistics

stat_daily_site(*start=None, end=None, attrs=['bytes', 'wan-tx_bytes', 'wan-rx_bytes', 'wlan_bytes', 'num_sta', 'lan-num_sta', 'wlan-num_sta', 'time']*)
Fetch site statistics with 1 day granularity

Parameters

- **start** (*int*) – *optional* start of reporting period, as seconds in the Unix epoch. If not present defaults to 1 year before the end time
- **end** (*int*) – *optional* end of reporting period, as seconds in the Unix epoch. If not present defaults to the current time.
- **attrs** (*list*) – *optional* list of statistics to return

Returns List of dictionaries of statistics

stat_5minutes_aps(*mac=None, start=None, end=None, attrs=['bytes', 'num_sta', 'time']*)
Fetch access point statistics with 5 minute granularity

Parameters

- **mac** (*str*) – *optional* MAC access of single AP for which to fetch statistics
- **start** (*int*) – *optional* start of reporting period, as seconds in the Unix epoch. If not present defaults to 12 hours before the end time
- **end** (*int*) – *optional* end of reporting period, as seconds in the Unix epoch. If not present defaults to the current time.
- **attrs** (*list*) – *optional* list of statistics to return

Returns List of dictionaries of statistics

stat_hourly_aps(*mac=None, start=None, end=None, attrs=['bytes', 'num_sta', 'time']*)
Fetch access point statistics with 1 hour granularity

Parameters

- **mac** (*str*) – *optional* MAC access of single AP for which to fetch statistics
- **start** (*int*) – *optional* start of reporting period, as seconds in the Unix epoch. If not present defaults to 7 yays before the end time
- **end** (*int*) – *optional* end of reporting period, as seconds in the Unix epoch. If not present defaults to the current time.
- **attrs** (*list*) – *optional* list of statistics to return

Returns List of dictionaries of statistics

stat_daily_aps(*mac=None, start=None, end=None, attrs=['bytes', 'num_sta', 'time']*)
Fetch access point statistics with 1 day granularity

Parameters

- **mac** (*str*) – *optional* MAC access of single AP for which to fetch statistics
- **start** (*int*) – *optional* start of reporting period, as seconds in the Unix epoch. If not present defaults to 1 year before the end time
- **end** (*int*) – *optional* end of reporting period, as seconds in the Unix epoch. If not present defaults to the current time.
- **attrs** (*list*) – *optional* list of statistics to return

Returns List of dictionaries of statistics

stat_5minutes_user(*mac, start=None, end=None, attrs=['time', 'rx_bytes', 'tx_bytes']*)
Fetch client device statistics with 5 minute granularity

Parameters

- **mac** (*str*) – MAC access of client device for which to fetch statistics
- **start** (*int*) – *optional* start of reporting period, as seconds in the Unix epoch. If not present defaults to 12 hours before the end time
- **end** (*int*) – *optional* end of reporting period, as seconds in the Unix epoch. If not present defaults to the current time.
- **attrs** (*list*) – *optional* list of statistics to return

Returns List of dictionaries of statistics

stat_hourly_user(*mac, start=None, end=None, attrs=['time', 'rx_bytes', 'tx_bytes']*)
Fetch client device statistics with 1 hour granularity

Parameters

- **mac** (*str*) – MAC access of client device for which to fetch statistics
- **start** (*int*) – *optional* start of reporting period, as seconds in the Unix epoch. If not present defaults to 7 days before the end time
- **end** (*int*) – *optional* end of reporting period, as seconds in the Unix epoch. If not present defaults to the current time.
- **attrs** (*list*) – *optional* list of statistics to return

Returns List of dictionaries of statistics

stat_daily_user(*mac, start=None, end=None, attrs=['time', 'rx_bytes', 'tx_bytes']*)
Fetch client device statistics with 1 day granularity

Parameters

- **mac** (*str*) – MAC access of client device for which to fetch statistics
- **start** (*int*) – *optional* start of reporting period, as seconds in the Unix epoch. If not present defaults to 1 year before the end time
- **end** (*int*) – *optional* end of reporting period, as seconds in the Unix epoch. If not present defaults to the current time.
- **attrs** (*list*) – *optional* list of statistics to return

Returns List of dictionaries of statistics

stat_sessions(*mac=None*, *start=None*, *end=None*, *type='all'*)
Show login sessions

stat_sta_sessions_latest(*mac*, *limit=5*)
Show latest 'n' login sessions for a single client device

stat_auths(*start=None*, *end=None*)
Show all authorizations

list_allusers(*type='all'*, *conn='all'*, *within=8760*)
List all client devices ever connected to the site

list_guests(*within=8760*)
List guest devices

list_clients(*client_mac=None*)
List currently connected client devices, or details on a single MAC address

list_configured_clients(*client_mac=None*)
List configured client devices, or details on a single MAC address

get_client_details(*client_mac*)
Get details about a client

list_usergroups()
List user groups

set_usergroup(*client_mac*, *usergroup_id*)
Set the user group for a client

edit_usergroup(*group_id*, *site_id*, *name*, *qos_rate_max_down=-1*, *qos_rate_max_up=-1*)
Update user group

create_usergroup(*name*, *qos_rate_max_down=-1*, *qos_rate_max_up=-1*)
Create user group

delete_usergroup(*group_id*)
Delete user group

list_health()
List health metrics

list_dashboard()
List dashboard metrics

list_users()
List knows clients groups

list_devices(*device_mac=None*)
List details of one or more managed device on this site

Parameters **device_mac** (*str*) – *optional* MAC address of device on which to fetch details

Returns list of dictionaries of device details.

list_devices_basic()
List basic information about managed devices

list_tags()
List known device tags

list_rogueaps(*within=24*)
List rogue or nearby APs

list_rogueknown()
List rogue or nearby APs

list_sites()
List sites on this controller

stat_sites()
Get stats for sites on this controller

create_site(*desc*)
Create a site

delete_site(*site*)
Delete a site

set_site_name(*desc*)
Change a site's name

set_site_country(*setting=None*)
Set site country

set_site_locale(*setting=None*)
Set site locale

set_site_snmp(*setting=None*)
Set site snmp

set_site_mgmt(*setting=None*)
Set site mgmt

set_site_guest_access(*setting=None*)
Set site guest access

set_site_ntp(*setting=None*)
Set site ntp

set_site_connectivity(*setting=None*)
Set site connectivity

list_admins()
List admins

list_all_admins()
List all admins

invite_admin(*name*, *email*, *readonly=False*, *enable_sso=True*, *device_adopt=False*, *device_restart=False*)
Invite a new admin for access to the current site

create_admin(*name*, *email*, *password*, *requires_new_password=False*, *readonly=False*, *enable_sso=True*, *device_adopt=False*, *device_restart=False*)
Create a new admin user

revoke_admin(*admin*)
Revoke an admin user

list_wlan_groups()
List wlan_groups

stat_sysinfo()
Show general system information

stat_status()
Get controller status

list_self()
Get info about the logged in user

list_vouchers(*create_time*)
List vouchers

stat_payment()
List payments

create_hotspotop(*name*, *password*, *note*)
Create hotspot operator

list_hotspotop()
List hotspot operators

create_voucher(*minutes*, *count=1*, *quota=0*, *note=None*, *up=None*, *down=None*, *MBytes=None*)
Create voucher(s)

revoke_voucher(*voucher_id*)
Revoke voucher

extend_guest_validity(*guest_id*)
Extend guest validity

list_portforward_stats()
List port forwarding configuation and statistics

list_vpn_stats()
List VPN users and statistics

list_dpi_stats()
List deep packet inspection stats

list_current_channels()
List currently available channels

list_country_codes()
List country codes

list_portforwarding()
List port forwarding settings

list_dynamicdns()
List dynamic DNS settings

list_portconf()
List port configurations

list_extension()
List VoIP extensions

list_settings(*key=None*)
List site settings

adopt_device(*mac*)
Adopt a device to the selected site

restart_ap(*mac*)
Reboot an access point

disable_ap(*ap_id*, *disabled*)
Disable/enable an access point

led_override(*device_id*, *led_override*)
Override LED mode for a device

locate_ap(*mac*, *enabled*)
Toggle flashing LED of an access point for locating purposes

site_leds(*led_enabled*)
Toggle LEDs of all the access points ON or OFF

set_ap_radiosettings(*ap_id*, *radio_table*, *channel*, *ht*, *tx_power_mode*, *tx_power*)
Update access point radio settings

rename_ap(*ap_id*, *name*)
Rename access point

move_device(*site*, *mac*)
Move a device to another site

delete_device(*mac*)
Delete a device from the current site

list_networkconf()
List network settings

create_network(*settings=None*)
Create a network

set_networksettings(*network_id*, *settings=None*)
Update network settings, base

delete_network(*network_id*)
Delete a network

list_wlanconf(*wlan_id=None*)
List wireless LAN settings for all or one network

set_wlan_settings(*wlan_id*, *passphrase*, *ssid=None*)
Set wireless LAN password and SSID

enable_wlan(*wlan_id*, *enabled*)
Enable or diabble a wireless LAN

set_wlan_mac_filter(*wlan_id*, *enabled*, *whitelist=False*, *mac_list=None*)
Set wireless LAN MAC filtering policy

delete_wlan(*wlan_id*)
Delete a wlan

list_events(*historyhours=720*, *start=0*, *limit=1000*)
List events

Parameters

- **historyhours** (*int*) – how far back to list events
- **start** (*int*) – index of the first event to return
- **limit** (*int*) – maximum number of events to return

list_alarms()
List all alarms

count_alarms()
Count alarms

archive_alarm(*alarm_id*)
Archive a single alarm

archive_all_alarms()
Archive alarms(s)

upgrade_device(*mac*)
Upgrade a device to the latest firmware

upgrade_device_external(*mac*, *url*)
Upgrade a device to a specific firmware file

start_rolling_upgrade()
Start rolling upgrade

cancel_rolling_upgrade()
Cancel rolling upgrade

power_cycle_switch_port(*mac*, *port_idx*)
Power-cycle the PoE output of a switch port

spectrum_scan(*mac*)
Trigger an RF scan by an AP

spectrum_scan_state(*ap_mac*)
Check the RF scanning state of an AP

set_device_settings_base(*device_id*, *settings=None*)
Update device settings, base

list_radius_profiles()
List Radius profiles

list_radius_accounts()
List Radius user accounts

create_radius_account(*name*, *password*, *tunnel_type*, *tunnel_medium_type*, *vlan=None*)
Create a Radius user account

set_radius_account_base(*account_id*, *account_details=None*)
Update Radius account, base

delete_radius_account(*account_id*)
Delete a Radius account

2.3 Constants

class RadiusTunnelType
Values for the tunnel type in RADIUS profiles

PPTP = 1
Point-to-Point Tunneling Protocol

L2F = 2
Layer Two Forwarding

L2TP = 3
Layer Two Tunneling Protocol

ATMP = 4
Ascend Tunnel Management Protocol

VTP = 5
Virtual Tunneling Protocol

AH = 6
IP Authentication Header in the Tunnel-mode

IP_IP = 7
IP-in-IP Encapsulation

MIN_IP_IP = 8
Minimal IP-in-IP Encapsulation

ESP = 9
IP Encapsulating Security Payload in the Tunnel-mode

GRE = 10
Generic Route Encapsulation

DVS = 11
Bay Dial Virtual Services

IP_IP_TUN = 12
IP-in-IP Tunneling

VLAN = 13
Virtual LANs

class RadiusTunnelMediumType
Values for the tunnel medium type in RADIUS profiles

IPv4 = 1
IP version 4

IPv6 = 2
IP version 6

NSAP = 3
NSAP

HDLC = 4
8-bit multidrop

BBN = 5
1822

IEEE_802 = 6
includes all 802 media plus Ethernet “canonical format”

E_163 = 7
E.163 (POTS)

E_164 = 8
E.164 (SMDS, Frame Relay, ATM)

F_69 = 9
F.69 (Telex)

X_121 = 10
X.121 (X.25, Frame Relay)

IPX = 11
IPX

APPLETALK = 12
Appletalk

DECNET = 13
Decnet IV

BANYAN = 14
Banyan Vines

E_164_NSAP = 15
E.164 with NSAP format subaddress

CHAPTER 3

SSL Security with self-signed certificates

The Unifi controller is accessed using the *https:* protocol in order protect the session. Unfortunately the way that they do this does not protect against *Man-In-The-Middle* attacks due to the use of a *self-signed certificate*. To understand why this is an issue and how to fix it it is necessary to understand a bit about what SSL certificates do and how they do it.

The SSL protocol (and its more modern successor, the TLS protocol) make use of *digital certificates*. These are essentially messages that are a digitally *signed* message that is *signed* by some party to state that a particular identity is connected to a particular *public key*. A *public key* is a value that can be used to verify a *digital signature* such as the ones on these certificates. Each certificate has an *issuer*, the party signing the message, and a *subject*, the party that is having its identity/key relationship asserted in this certificate. In order to validate a certificate you need to have a copy of the *public key* associated with the *issuer*. The *public key* belonging to the *subject* of a certificate sent in the course of starting an SSL session is used to validate *digital signatures* in the SSL handshake messages and this is used as evidence that the server with which you are communicating belongs to the *subject* of the certificate.

When you make an SSL connection on the internet it is typical for the server at the other end to have a *certificate* issued by some well know authority. Your web browser has the public keys of many well know authorities built in to it. In these sorts of certificate the identity of the *subject* includes the domain name of the server to which you are connecting and these authorities are supposed to only issue certificate to the owners of the domains. This way you can have confidence that you are connecting to the right server and not to some system that is trying to eavesdrop on your conversation.

The Unifi controller (and many other local servers and appliances) typically does not have a public, externally accessible domain name and even if it did, getting a certificate for that domain name is often time consuming and expensive. As a result what Ubiquiti (along with most appliance vendors) does is create a *self-signed certificate*. This is a certificate for which the *issuer* is not some well known authority but is instead the same identity as the *subject*. The first time you fire up the Unifi controller it spots that it doesn't have a certificate and creates a new one, signed by itself, and identifying the server with the host name *unifi*.

There are two problems with this approach. Firstly, since the *issuer* of the certificate is not a well known authority many systems will complain that the certificate is issued by an unknown party. Secondly, unless you access your Unifi controller using the unqualified domain name *unifi* the host name in the certificate will not match the host name used to access the server, and again the system will complain about a mismatched domain name. Furthermore, since the certificate was just created out of thin air, if you anticipate and ignore these two warnings then there is nothing to stop

an eavesdropper from simply creating a new *self-signed certificate* and fooling you into sending your credentials to a bogus server instead of the Unifi controller.

Fortunately there is a solution to these problems. The solution is known as *certificate pinning*. This basically just means that you expect to see the same certificate every time you access the same server. This won't help if the eavesdropper is already intercepting your connections the first time you access a service but it will protect you for all subsequent accesses.

This library implements certificate pinning.

CHAPTER 4

Extending the Unifi API

Ubiquiti are constantly enhancing the Unifi controller and each new release adds new functionality. As such new functionality will need to be added to this library as time goes by. Part of the design goal of this library was to make the addition of new API calls as simple as possible.

4.1 Metaprogramming

The main `unificontrol.UnifiClient` class implements the more than 100 API calls to read and change settings on the controller. The vast majority of these calls map directly to a single https access to a specific endpoint on the Unifi controller web service. In order to avoid a great deal of repetition and boilerplate code each of these calls is created using metaprogramming; rather than writing code to implement each one, the functions are instead described at a high level and the details are created when the class is first loaded.

There are several advantages to using metaprogramming in this stiuation. Chief among these are:

- The nature and the intent of the function are easier to see, since there is less extraneous text.
- There is less code overall, which reduces the space for bugs to creep in (and also reduces finger fatigue).
- There is a great deal less repetition, which makes refactoring easier.
- Separating the specification from the implementation makes it easier to change either one.

Of course all of these aid with the main goal of making it easy to add new API calls when the controller gets enhancements.

4.2 Implementing API calls

Most of the API calls in the `unificontrol.UnifiClient` class are implemented simply by passing a description of the API call to an internal function called *UnifiAPICall* which constructs the necessary function call. For example the `list_alarms` method is implemented with the following code:

```
list_alarms = UnifiAPICall(
    "List all alarms",
    "list/alarm",
    )
```

In this example we only pass the two required parameters to *UnifiAPICall*, a documentation string and part of the path to the HTTP endpoint for th API call on the server. Of course for many API calls there are parameters that need to be passed. For instance, you can fetch details about managed Unifi devices using the *list_devices* method and in this case you may optionally specify the MAC address of the managed device on the URL used to connect to the controller. When that is the case we can specify a name to give to a parameter for the extra componenet to be added to the URL in this case `device_mac`:

```
list_devices = UnifiAPICall(
    """List details of one or more managed device on this site

    Args:
        device_mac (str): `optional` MAC address of device on which to fetch details

    Returns:
        list of dictionaries of device details.
    """,
    "stat/device",
    path_arg_name="device_mac",
    )
```

Often we want to pass a bunch of setting to the controller and these are usually sent by POSTing a JSON object containing the settings. Consider the case of the *edit_usergroup* method:

```
edit_usergroup = UnifiAPICall(
    "Update user group",
    "rest/usergroup",
    path_arg_name="group_id",
    path_arg_optional=False,
    json_args=['site_id',
               'name',
               ('qos_rate_max_down', -1),
               ('qos_rate_max_up', -1)],
    method="PUT",
    )
```

Here the use must specify the `group_id` that is being edited (and since this is a requirement we set `path_arg_optional` to `False` to ensure that the user knows it's required). We also need to pass some arguments in the JSON object to set the `site_id`, the `name` of the group and optionally bandwidth limits for upstream and downstream traffic. These are descibed in the `json_args` list; the first two (required) entries justy have names but for the last two we pass a tuple of `(name, default)` (the controller interprets the a value of -1 for either of these last two as unlimited). In this example we also see that this endpoint expects the configuration to be delivered in an HTTP PUT, rather than a POST, so we also provide a `method` value.

In some cases an HTTP endpoint is used to implement multiple operations, in which case the operation itself is also specified in the JSON payload. In this case you need to also need to specify the `rest_command` that will be passed as part of the JSON payload:

```
revoke_admin = UnifiAPICall(
    "Revoke an admin user",
    "cmd/sitemgr",
    rest_command="revoke-admin",
```

(continues on next page)

(continued from previous page)

```
    json_args=['admin'],
    )
```

Sometimes the raw JSON arguments expected by the controller have names that are not very descriptive. Sometimes they take only certain values and it would be helpful to do some value checking. Sometimes we would like to pass default values that are not constants but are more context-sensitive. Sometimes we want to set hidden parameters based on the specified parameters. In all of these cases what we really need to do is filter the JSON arguments dictionary before we pass it to the controller. To do this we can use the `json_fix` argument. For example:

```
invite_admin = UnifiAPICall(
    "Invite a new admin for access to the current site",
    "cmd/sitemgr",
    json_args=['name',
               'email',
               ('readonly', False),
               ('enable_sso', True),
               ('device_adopt', False),
               ('device_restart', False)],
    rest_command='invite-admin',
    json_fix=[fix_arg_names({'enable_sso':'for_sso'}),
              fix_admin_permissions,
              fix_check_email('email')],
    )
```

Here we apply several fixer functions (in order). The first renames the argument `enable_sso` to the slightly more esoteric internal name `for_sso`, the second converts some flag paramters to an internal dictionary representation used for the admin permissions and the third ensures that the `email` parameter contains a valid email address.

See the *JSON fix-up methods* section for a list of the current JSON fix-up functions.

For some of the operations, particularly for setting site-wide and network-specific settings, it makes more sense for the Python API to accept a dictionary of values to pass as the JSON request body rather than taking a large number of method arguments. In this case you can use the `json_body_name` arguement to set the name of the method argument under which this JSON value will be provided to the API.

```
set_site_snmp = UnifiAPICall(
    "Set site snmp",
    "rest/setting/snmp",
    json_body_name="setting",
    method="PUT",
    )
```

Most of the calls in the API apply to the settings for just one of the sites under management but a few apply to the controller as a whole. In these cases the method is created using `UnifiAPICallNoSite` instead of `UnifiAPICall`. Also, so the few calls that do not require the user to be logged in you may pass `need_login=False` to indicate that the client object does not need to automatically log the user in and authentication failures should not trigger a login attempt.

4.3 JSON fix-up methods

Functions in the `unificontrol.json_fixers` module are fixers to fix up JSON objects before posting to the controller. This allows us to have cleaner function signatures when the underlying API is a bit verbose.

All functions accept a JSON dictionary of existing attributes and return a modified dictionary, which may or may not be the same object.

fix_arg_names (*mapping*)
Given a mapping, return a fixer that renames the json arguments listed in the mapping. For example:

```
json_fix = [fix_arg_names({'enable_sso':'for_sso'})] # Let the user write␣
↪``enable_sso`` when the API wants ``for_sso``
```

fix_check_email (*field_name*)
Given the name of a field return a fixer that check that that field is a valid email address

fix_constants (*constants*)
Given a dict of constant parameters this function returns a fixer function that updates the json to include these constants

fix_end_now (*json*)
Set end time to the time now if no end time is give

fix_enforce_values (*mapping*)
Given a mapping create a fixer that checks the value in an argument and raises a (helpful) ValueError exception if the value is not one listed

fix_ensure_time_attrib (*json*)
Ensure that requested attributes include the 'time' attribute

fix_macs_list (*json*)
Convert a single mac into a list as necessary

fix_note_noted (*json*)
Ensure the 'noted' flag is set if and only if a note is given

fix_start_12hours (*json*)
Fix start to 12 hours before end if not given

fix_start_1year (*json*)
Fix start to 1 year before end if not given

fix_start_7days (*json*)
Fix start to 7 days before end if not given

fix_times_as_ms (*json*)
Adjust start and end times to be in milliseconds rather than seconds

CHAPTER 5

Indices and tables

- genindex
- search

Python Module Index

u

unificontrol.json_fixers, 21

Index

A

adopt_device() (*UnifiClient method*), 13
AH (*RadiusTunnelType attribute*), 15
APPLETALK (*RadiusTunnelMediumType attribute*), 16
archive_alarm() (*UnifiClient method*), 14
archive_all_alarms() (*UnifiClient method*), 14
ATMP (*RadiusTunnelType attribute*), 15
authorize_guest() (*UnifiClient method*), 6

B

BANYAN (*RadiusTunnelMediumType attribute*), 16
BBN (*RadiusTunnelMediumType attribute*), 15
block_client() (*UnifiClient method*), 7

C

cancel_rolling_upgrade() (*UnifiClient method*), 14
count_alarms() (*UnifiClient method*), 14
create_admin() (*UnifiClient method*), 11
create_client() (*UnifiClient method*), 7
create_hotspotop() (*UnifiClient method*), 12
create_network() (*UnifiClient method*), 13
create_radius_account() (*UnifiClient method*), 14
create_site() (*UnifiClient method*), 11
create_usergroup() (*UnifiClient method*), 10
create_voucher() (*UnifiClient method*), 12

D

DECNET (*RadiusTunnelMediumType attribute*), 16
delete_device() (*UnifiClient method*), 13
delete_network() (*UnifiClient method*), 13
delete_radius_account() (*UnifiClient method*), 14
delete_site() (*UnifiClient method*), 11
delete_usergroup() (*UnifiClient method*), 10
delete_wlan() (*UnifiClient method*), 13
disable_ap() (*UnifiClient method*), 13
DVS (*RadiusTunnelType attribute*), 15

E

E_163 (*RadiusTunnelMediumType attribute*), 16
E_164 (*RadiusTunnelMediumType attribute*), 16
E_164_NSAP (*RadiusTunnelMediumType attribute*), 16
edit_usergroup() (*UnifiClient method*), 10
enable_wlan() (*UnifiClient method*), 13
ESP (*RadiusTunnelType attribute*), 15
extend_guest_validity() (*UnifiClient method*), 12

F

F_69 (*RadiusTunnelMediumType attribute*), 16
fix_arg_names() (*in module unificontrol.json_fixers*), 21
fix_check_email() (*in module unificontrol.json_fixers*), 22
fix_constants() (*in module unificontrol.json_fixers*), 22
fix_end_now() (*in module unificontrol.json_fixers*), 22
fix_enforce_values() (*in module unificontrol.json_fixers*), 22
fix_ensure_time_attrib() (*in module unificontrol.json_fixers*), 22
fix_macs_list() (*in module unificontrol.json_fixers*), 22
fix_note_noted() (*in module unificontrol.json_fixers*), 22
fix_start_12hours() (*in module unificontrol.json_fixers*), 22
fix_start_1year() (*in module unificontrol.json_fixers*), 22
fix_start_7days() (*in module unificontrol.json_fixers*), 22
fix_times_as_ms() (*in module unificontrol.json_fixers*), 22
forget_client() (*UnifiClient method*), 7

G

get_client_details() (*UnifiClient method*), 10

GRE (*RadiusTunnelType attribute*), 15

H

HDLC (*RadiusTunnelMediumType attribute*), 15
host (*UnifiClient attribute*), 6

I

IEEE_802 (*RadiusTunnelMediumType attribute*), 15
invite_admin() (*UnifiClient method*), 11
IP_IP (*RadiusTunnelType attribute*), 15
IP_IP_TUN (*RadiusTunnelType attribute*), 15
IPv4 (*RadiusTunnelMediumType attribute*), 15
IPv6 (*RadiusTunnelMediumType attribute*), 15
IPX (*RadiusTunnelMediumType attribute*), 16

L

L2F (*RadiusTunnelType attribute*), 15
L2TP (*RadiusTunnelType attribute*), 15
led_override() (*UnifiClient method*), 13
list_admins() (*UnifiClient method*), 11
list_alarms() (*UnifiClient method*), 14
list_all_admins() (*UnifiClient method*), 11
list_allusers() (*UnifiClient method*), 10
list_clients() (*UnifiClient method*), 10
list_configured_clients() (*UnifiClient method*), 10
list_country_codes() (*UnifiClient method*), 12
list_current_channels() (*UnifiClient method*), 12
list_dashboard() (*UnifiClient method*), 10
list_devices() (*UnifiClient method*), 10
list_devices_basic() (*UnifiClient method*), 11
list_dpi_stats() (*UnifiClient method*), 12
list_dynamicdns() (*UnifiClient method*), 12
list_events() (*UnifiClient method*), 13
list_extension() (*UnifiClient method*), 12
list_guests() (*UnifiClient method*), 10
list_health() (*UnifiClient method*), 10
list_hotspotop() (*UnifiClient method*), 12
list_networkconf() (*UnifiClient method*), 13
list_portconf() (*UnifiClient method*), 12
list_portforward_stats() (*UnifiClient method*), 12
list_portforwarding() (*UnifiClient method*), 12
list_radius_accounts() (*UnifiClient method*), 14
list_radius_profiles() (*UnifiClient method*), 14
list_rogueaps() (*UnifiClient method*), 11
list_rogueknown() (*UnifiClient method*), 11
list_self() (*UnifiClient method*), 12
list_settings() (*UnifiClient method*), 13
list_sites() (*UnifiClient method*), 11
list_tags() (*UnifiClient method*), 11
list_usergroups() (*UnifiClient method*), 10
list_users() (*UnifiClient method*), 10
list_vouchers() (*UnifiClient method*), 12
list_vpn_stats() (*UnifiClient method*), 12
list_wlan_groups() (*UnifiClient method*), 12
list_wlanconf() (*UnifiClient method*), 13
locate_ap() (*UnifiClient method*), 13
login() (*UnifiClient method*), 6
logout() (*UnifiClient method*), 6

M

MIN_IP_IP (*RadiusTunnelType attribute*), 15
move_device() (*UnifiClient method*), 13

N

NSAP (*RadiusTunnelMediumType attribute*), 15

P

port (*UnifiClient attribute*), 6
power_cycle_switch_port() (*UnifiClient method*), 14
PPTP (*RadiusTunnelType attribute*), 15

R

RadiusTunnelMediumType (*class in unificontrol*), 15
RadiusTunnelType (*class in unificontrol*), 15
reconnect_client() (*UnifiClient method*), 7
rename_ap() (*UnifiClient method*), 13
restart_ap() (*UnifiClient method*), 13
revoke_admin() (*UnifiClient method*), 12
revoke_voucher() (*UnifiClient method*), 12

S

set_ap_radiosettings() (*UnifiClient method*), 13
set_client_fixed_ip() (*UnifiClient method*), 7
set_client_name() (*UnifiClient method*), 7
set_client_note() (*UnifiClient method*), 7
set_device_settings_base() (*UnifiClient method*), 14
set_networksettings() (*UnifiClient method*), 13
set_radius_account_base() (*UnifiClient method*), 14
set_site_connectivity() (*UnifiClient method*), 11
set_site_country() (*UnifiClient method*), 11
set_site_guest_access() (*UnifiClient method*), 11
set_site_locale() (*UnifiClient method*), 11
set_site_mgmt() (*UnifiClient method*), 11
set_site_name() (*UnifiClient method*), 11
set_site_ntp() (*UnifiClient method*), 11

set_site_snmp() (*UnifiClient method*), 11
set_usergroup() (*UnifiClient method*), 10
set_wlan_mac_filter() (*UnifiClient method*), 13
set_wlan_settings() (*UnifiClient method*), 13
site (*UnifiClient attribute*), 6
site_leds() (*UnifiClient method*), 13
spectrum_scan() (*UnifiClient method*), 14
spectrum_scan_state() (*UnifiClient method*), 14
start_rolling_upgrade() (*UnifiClient method*), 14
stat_5minutes_aps() (*UnifiClient method*), 8
stat_5minutes_site() (*UnifiClient method*), 8
stat_5minutes_user() (*UnifiClient method*), 9
stat_auths() (*UnifiClient method*), 10
stat_daily_aps() (*UnifiClient method*), 9
stat_daily_site() (*UnifiClient method*), 8
stat_daily_user() (*UnifiClient method*), 9
stat_hourly_aps() (*UnifiClient method*), 8
stat_hourly_site() (*UnifiClient method*), 8
stat_hourly_user() (*UnifiClient method*), 9
stat_payment() (*UnifiClient method*), 12
stat_sessions() (*UnifiClient method*), 10
stat_sites() (*UnifiClient method*), 11
stat_sta_sessions_latest() (*UnifiClient method*), 10
stat_status() (*UnifiClient method*), 12
stat_sysinfo() (*UnifiClient method*), 12

U

unauthorize_guest() (*UnifiClient method*), 6
unblock_client() (*UnifiClient method*), 7
UnifiClient (*class in unificontrol*), 6
unificontrol.json_fixers (*module*), 21
upgrade_device() (*UnifiClient method*), 14
upgrade_device_external() (*UnifiClient method*), 14

V

VLAN (*RadiusTunnelType attribute*), 15
VTP (*RadiusTunnelType attribute*), 15

X

X_121 (*RadiusTunnelMediumType attribute*), 16