sales@candelatech.com
support@candelatech.com
+1 (360) 380-1618 [PST, GMT -8]

Create Python Scripts Utilizing the Realm Library

Goal: Create a python script to create stations and Layer-3 cross connects
Using the `realm.py` library we will write a script that will allow us to automate the creation of stations and Layer-3 cross connects. We will also be able to start and stop traffic over the cross connects using the script. We will be referencing the script, test_ipv4_variable_time.py, as an example throughout this cookbook. Requires LANforge 5.4.2.

1. Starting the script

A. Setting up inheritance for our object

A. In order for our script to be platform independent we will need to `import sys`. Then use

```
if 'py-json' not in sys.path:
    sys.path.append(os.path.join(os.path.abspath('..'), 'py-json'))
```

B. When creating our object we will need to import the LFCliBase module from the LANforge module using `from LANforge.lfcli_base import LFCliBase`

C. After importing LFCliBase we can create our Class and inherit from LFCliBase

B. Setting up the main method

A. The main method will typically follow a pattern:

I. First, the creation of a list of stations. This can be done in many ways.
Example:

```
station_list = LFUtils.port_name_series(prefix_="sta",
    start_id_=0,
    end_id_=4,
    padding_number_=10000)
```

II. Following the station list, we can initialize our object:

```
ip_var_test = IPV4VariableTime(lfjson_host, lfjson_port,
    number_template="00",
    sta_list=station_list,
    name_prefix="var_time",
    ssid="testNet",
    password="testPass",
    resource=1,
    security="wpa2",
    test_duration="5m",
    side_a_min_rate=256,
    side_b_min_rate=256)
```

III. After our object has been initialized we can begin the testing process. The preferred order for running our tests is to:

i. Call `cleanup()` to prevent stations, cross-connects, and endpoints within our list from having creation issues if anything exists with the same name.

ii. Call the `build()` method in our class to setup the basic versions of the stations, cross-connects, and endpoints.

iii. Call the `start()` method that will start the test itself, as well as any bring up any stations and start traffic on cross-connects that need it.

iv. Call the `stop()` method to stop the traffic and bring down any stations that are up.

v. Verify that the tests passed using our inherited passes() method.

vi. After verifying a pass we can then call our cleanup function again to clean up everything we worked with.

C. Example Main Method

```
def main():
    lfjson_host = "localhost"
    lfjson_port = 8080
    station_list = LFUtils.portNameSeries(prefix_="sta", start_id_=0, end_id_=4, paddin
    ip_var_test = IPV4VariableTime(lfjson_host, lfjson_port, number_template="00", sta_
        name_prefix="var_time",
        ssid="testNet",
        password="testPass",
        resource=1,
        security="wpa2", test_duration="5m",
        side_a_min_rate=256, side_b_min_rate=256),
    ip_var_test.cleanup(station_list)
    ip_var_test.build()
    if not ip_var_test.passes():
        print(ip_var_test.get_fail_message())
        exit(1)
    ip_var_test.start(False, False)
    ip_var_test.stop()
    if not ip_var_test.passes():
        print(ip_var_test.get_fail_message())
        exit(1)
    time.sleep(30)
    ip_var_test.cleanup(station_list)
    if ip_var_test.passes():
        print("Full test passed, all connections increased rx bytes")
```

2. ## Test Methods Available With Realm

A. Using `lfcli_base._pass()` and `lfcli_base._fail()`

A. Since our class is inheriting lfcli_base.py, we have access to methods that will help us keep track of passes and fails during our tests. We can access them using `self._pass()` or `self._fail()`. They will take two parameters, a string `message` and an optional boolean `print_pass` and `print_fail` for `_pass()` and `_fail()` respectively. If `print_pass` or `print_fail` are set to True, they will write the message to stdout whenever the functions are called.

B. `lfcli_base` will add a "PASSED: *message*" or "FAILED: *message*" to a list when the tests pass or fail. This list can be accessed using the methods

```
get_result_list()
get_failed_result_list()
get_fail_message()
get_all_message()
```

B. Using lfcli_base to check test success

A. `passes()` will return a boolean depending on whether or not there were any fails in the test. If it finds a fail message it will return False, if none are found it will return True.
`get_result_list()` will return all logged pass/fail messages as a list.
`get_failed_result_list()` will return a list of only fail messages.
`get_fail_message()` will return a list of string of fail messages separated by newlines
`get_message()` will return a list of string of all messages separated by newlines

3. ## Building a Station

A. Build Method

A. We will need to do a number of things to setup our build method.

I. To begin we will set the security type of our stations using `station_profile.use_security()`

II. We will then use `station_profile.set_number_template()` to name our stations

III. After this we can set our command flags and parameters using

```
self.station_profile.set_command_flag("add_sta","create_admin_down",1)
self.station_profile.set_command_param("set_port","report_timer",1500)
self.station_profile.set_command_flag("set_port","rpt_timer", 1)
```

IV. Once our parameters and flags are set, we can pass a list of stations to `station_profile.create()` and `cx_profile.create()`. Our build function could look like this:

```
for station in range(len(self.sta_list)):
    temp_sta_list.append(str(self.resource)+"."+self.sta_list[station])
self.station_profile.create(resource=1, radio="wiphy0", sta_names_=self.sta_list, debug=Fal
self.cx_profile.create(endp_type="lf_udp", side_a=temp_sta_list, side_b="1.eth1", sleep_tim
self._pass("PASS: Station build finished")
```

i The naming convention for the sides will look like `foo-A` for side_a and `foo-B` for side_b. *foo* will be set based on the names in the list of stations given.

B. StationProfile

A. The preferred method for creating a station_profile is to use the factory method `new_station_profile()` found in realm

I. We will need to assign some variables for the creation of our stations before we can call `create()`.

i. `self.station_profile.use_security(`*`security_type`*`, `*`ssid`*`, `*`passwd`*`)` is the preferred method to use when setting the security type, ssid, and password variables
Example:

```
self.station_profile.use_security("wpa2", "testNet", "testPass")
```

ii. `self.station_profile.number_template_` is the numerical prefix for stations. Using a `number_template` of "00" will have stations look like *sta01, sta02...sta10*
Example:

```
self.station_profile.number_template_="00"
```

iii. `self.station_profile.mode` determines the wifi mode used by the stations. See here for available modes
Example:

```
self.station_profile.mode=0
```

4. # Cross Connects

A. Starting and Stopping Traffic

A. In order for us to be able to start traffic, our stations will need to be admined up, associated, and with an IP. We can bring them up using `station_profile.admin_up()`. We can then use `realm.wait_for_ip(resource, sta_list)` to wait for our stations, as well as eth1, to get an IP address.

B. Once we are sure all of our stations have ip addresses, we can use `cx_profile.start_cx()` to start the traffic for our cross-connects. When we decide to stop the traffic we can just as easily use `cx_profile.stop_cx()` to stop traffic.

B. L3CXProfile

A. `self.local_realm.create_new_l3_cx_profile()` is the preferred method for creating a new Layer 3 CX Profile.

I. We will need to assign some variables for the creation of our stations before we can call `create()`.

i. `self.cx_profile.name_prefix` will be used to specify the name prefix for the cx. Assigning `self.cx_profile.name_prefix` to "test_" would produce cross-connects named test_sta00 with the numbers being dependent on station_profile's `number_template`.
Example:
```
self.cx_profile.name_prefix="test_"
```

ii. Set the `_min_bps` to the desired amount. `_max_bps` can be set but typically defaults to 0 which sets it to the same as the minimum bps.
Example:
```
self.cx_profile.side_a_min_bps=56000
self.cx_profile.side_b_min_bps=56000
```

5. ## Using TTLS

A. TTLS setup requires a few pieces of information to work correctly. `StationProfile` has a `set_wifi_extra()` method for setting the relevant variables. See here for the available options

B. We will need a key management type (`key_mgmt`), an EAP method (`eap`), an EAP identity string (`identity`), an EAP password string (`passwd`), an 802.11u realm (`realm`), an 802.11u domain (`domain`), and an 802.11u HESSID (`hessid`)
Example:
```
key_mgmt="WPA-EAP"
eap="TTLS"
identity="testuser"
passwd="testpasswd"
realm="localhost.localdomain"
domain="localhost.localdomain"
hessid="00:00:00:00:00:01"
```
We can then use these variables to call the `set_wifi_extra()` method
Example:
```
station_profile.set_wifi_extra(key_mgmt, eap, identity, passwd, realm, domain, hessid)
```

6. ## Cleaning Up

A. Cleanup stations and cross connects

A. We have two options for cleaning up everything once we finish:

I. The preferred method to cleanup is to use the individual cleanup methods found in `StationProfile` and `L3CXProfile`. These are `station_profile.cleanup(resource, desired_station_list)` and `cx_profile.cleanup()`. These methods are preferred because they will only delete stations, cross-connects, and endpoints created during the test while leaving others untouched.This is useful if you are running other scripts in the background.

II. The other method for cleanup is to use `Realm`'s `remove_all_stations(), remove_all_endps(), and remove_all_cxs()` methods. These will remove all stations, cxs, and endpoints that exist. These are good for doing a full cleanup, and it is recommended to use them in the order of cx, endpoint, station to prevent potential issues or missed deletions.

7. ## Debugging Stations

A. Debug information for station creation can be output by setting `_debug_on=True` in `StationProfile.create()`

   A. There are a few important debug outputs to pay attention to:

      I. This is the debug output that appears when using the add_sta command. This is used frequently in `StationProfile.create()`. This debug output will allow you to troubleshoot any **flags** or other information that is being set when creating your stations. It will output the name at the top and the raw JSON data will follow.

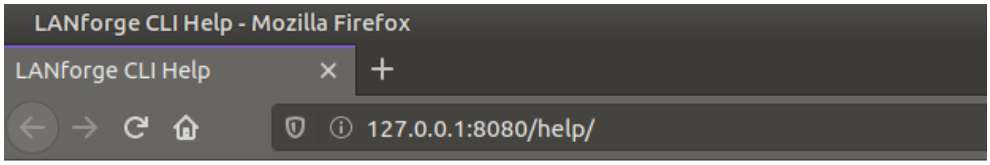
```
- 381 - sta0000- - - - - - - - - - - - - - - - - -
{'flags': 132096,
'flags_mask': 68719608832,
'key': 'testPass',
'mac': 'xx:xx:xx:xx:*:xx',
'mode': 0,'
radio': 'wiphy0',
'resource': 1,
'shelf': 1,
'ssid': 'testNet,
'sta_name': 'sta0000'}
```

      II. The next bit of debugging output comes from using the set_port command. We are able to see all of the JSON data that is posted, and can use this to check our flags and other info.

```
{'current_flags': 2147483649,
'interest': 8437762,
'port': 'sta0000',
'report_timer': 1500,
'resource': 1,
'shelf': 1}
<LANforge.LFRequest.LFRequest object at 0x7f13dbc56850>
-~381 - - - - - - - - - - - - - - - - - - - -
```

B. There are a few steps we can take to make validating the information we get through debugging easier.

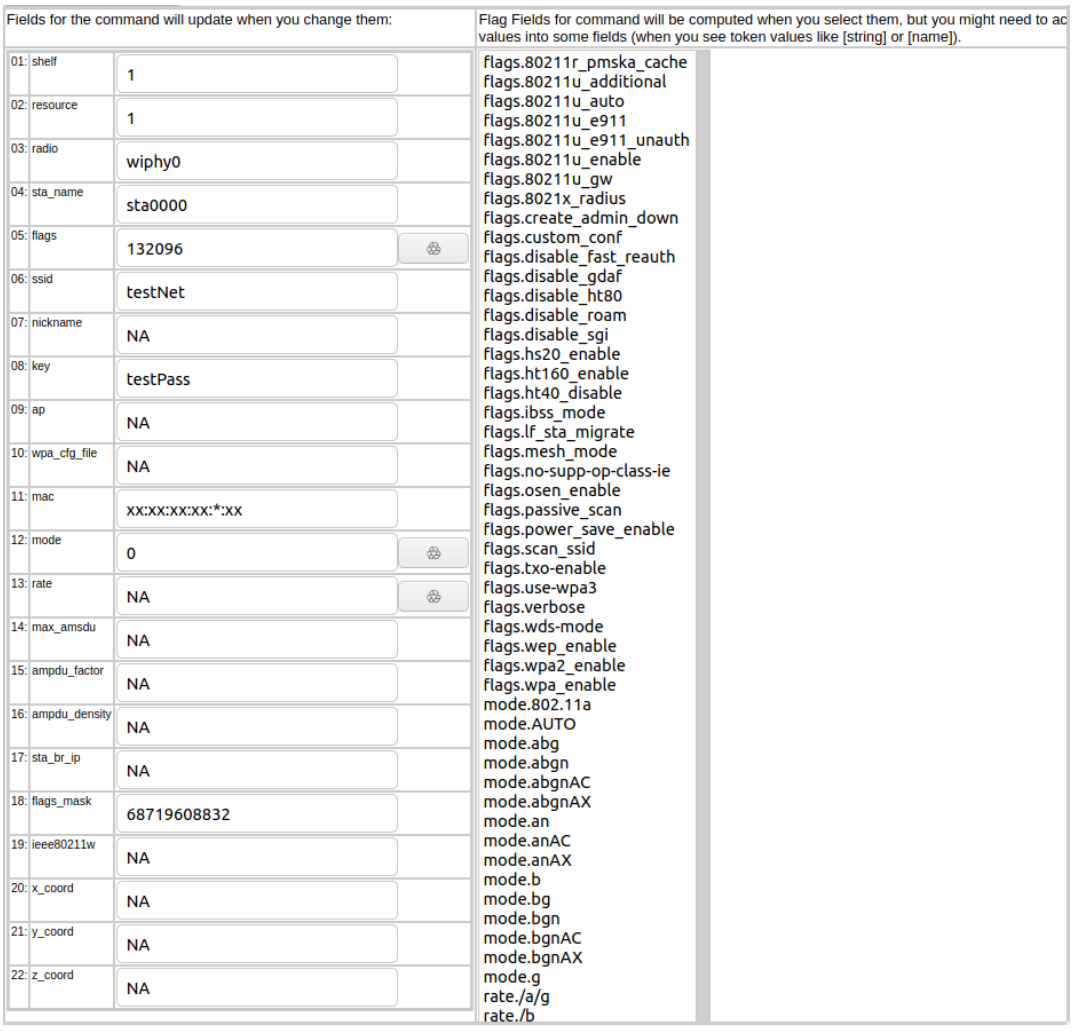
   A. We can use the help page available on the address of the machine LANforge is running on http://127.0.0.1/help/ will take us to a page containing all of the commands we can get help with.



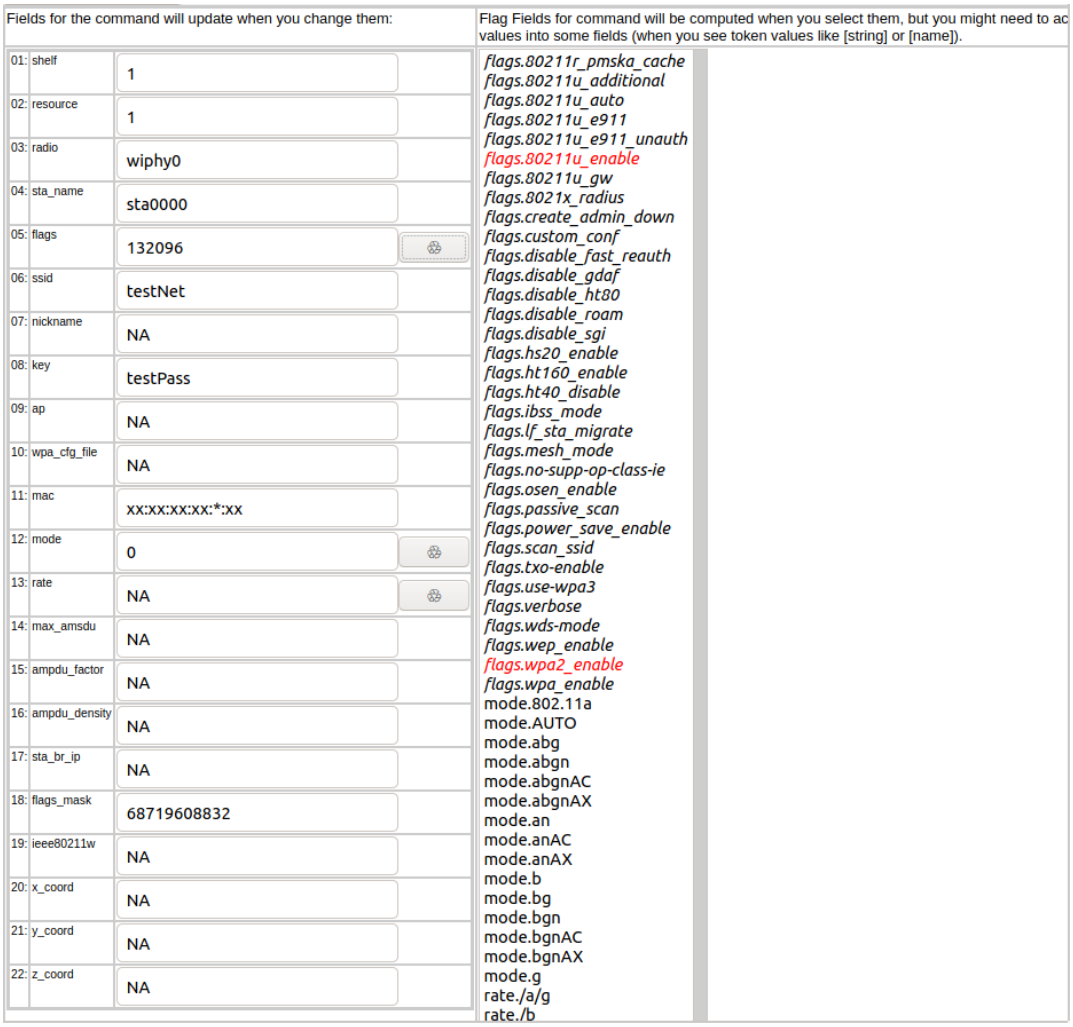
add_arm_endp	CLI Reference for /help/add_arm_endp
add_bgp_peer	CLI Reference for /help/add_bgp_peer
add_bond	CLI Reference for /help/add_bond
add_br	CLI Reference for /help/add_br
add_cd	CLI Reference for /help/add_cd
add_cd_endp	CLI Reference for /help/add_cd_endp
add_cd_vr	CLI Reference for /help/add_cd_vr
add_chamber	CLI Reference for /help/add_chamber
add_chamber_cx	CLI Reference for /help/add_chamber_cx
add_chamber_path	CLI Reference for /help/add_chamber_path
add_channel_group	CLI Reference for /help/add_channel_group
add_cx	CLI Reference for /help/add_cx
add_dut	CLI Reference for /help/add_dut
add_dut_notes	CLI Reference for /help/add_dut_notes
add_endp	CLI Reference for /help/add_endp
add_event	CLI Reference for /help/add_event

B. Using http://127.0.0.1/help/add_sta will bring us to a page specific to the `add_sta` command.

C. Here we can enter all of the data we got from our debugging output into the correct areas.



D. Flag fields have a button next to them that will calculate and highlight relevant flags in the right hand column of the page. This can be useful for checking that the correct flags are being set.



E. After we have done this, we can click the **parse command** button towards the top of the data inputs. We can then enter this command into LANforge's messages tab in the input box.