pyaoscx

Release 0.1.0

Mar 12, 2020

Contents:

The AOS-CX operating system provides a REST API to enable automated configuration and management of switches. This package contains Python modules that can be called upon to access the REST API and configure various features on the switches.

Note: Starting from AOS-CX release 10.04, there are two version of the REST API, the legacy v1 API and the newer v10.04 API.

CHAPTER 1

Getting Started

Important: This package is compatible with Python 3. Python 2 is not supported.

First ensure that REST API access is enabled and set to read/write mode with the following commands:

```
switch(config)# https-server rest access-mode read-write
switch(config)# https-server vrf mgmt
```

Additionally, ensure that the switch is reachable via an out-of-band management IP address with the appropriate login credentials.

1.1 Installation Instructions

Python comes with a package management system, pip. Pip can install, update, or remove packages from the Python environment. It can also do this for virtual environments. It is a good idea to create a separate virtual environment for this project. A guide to virtual environments can be found here.

To use pip to install the pyaoscx package from the official Python Package Index, PyPI, excecute the following command:

```
pip install pyaoscx
```

1.2 Package Structure

```
pyaoscx
|    README.md
|    Contributing.md
|    ...
```

(continues on next page)

(continued from previous page)

```
│
└──pyaoscx
│   │   access_security.py
│   │   acl.py
│   │   ...
│
└──docs
│   │   ...
│
└──workflows
    │   cleanup_l2_l3_vlans.py
    │   configure_l2_l3_vlans.py
    └──print_system_info.py
```

The `pyaoscx` package is a directory containing files and subdirectories. Contained directly within the top level `pyaoscx` directory are informative files such as the readme, licensing information, contribution guidelines, and release notes. Also contained within the pyaoscx package are the subdirectories relevant to the developer, `pyaoscx` and `workflows`.

1.2.1 pyaoscx

The `pyaoscx` subfolder contains the AOS-CX Python modules. Each module contains function definitions which either make a single REST API call or make multiple REST API calls. Each function that makes multiple REST API calls is written as such to perform a small logical low-level process (e.g. create a VLAN and then subsequently create a corresponding VLAN interface). The REST API calls are performed using the Python `requests` library, which provides functions to make HTTP GET, PUT, POST, and DELETE requests.

1.2.2 workflows

Workflows are scripts that combine calls to the functions in the `pyaoscx` modules (API call functions and low-level functions) to emulate larger network configuration processes. The `workflows` subfolder contains three example workflows. Each contains comments that describe step-by-step the operations being performed. You can make copies of these workflows in your own user directory and run them directly, or you can reference these examples to create your own workflows.

1.3 Executing Workflows

After installing the package, you should be able to execute workflows. Let's try that out by executing the simplest example workflow, print_system_info.py. This workflow simply logs into a switch and then prints switch configuration info to the console. To execute this workflow, first, copy this workflow from the workflows folder to a working directory outside of the package. Open your copy of the script and you'll notice that the `try:` block in the `main()` function consists of these statements:

```
session_dict = dict(s=session.login(base_url, username, password), url=base_url)

    system_info_dict = system.get_system_info(params={"selector": "configuration"},␣
↪**session_dict)

    pprint.pprint(system_info_dict)
```

The first two lines call the `session.login()` and `system.get_system_info()` functions in the `session` and `system` modules, respectively. The third line simply prints the system information out to the console in readable format.

Execute the workflow. You'll be asked to input the switch's out-of-band management IP address, login username, and password. After you do that successfully, the rest of the script should run and you should see this output:

```
C:\Users\wangder\AppData\Local\Programs\Python\Python37-32\python.exe "C:/Users/
↪wangder/OneDrive - Hewlett Packard Enterprise/git/packaging/pyaoscx/workflows/print_
↪system_info.py"
Switch IP Address: <IP address redacted>
Switch login username: <username redacted>
Switch login password: <password redacted>
INFO:root:SUCCESS: Login succeeded
INFO:root:SUCCESS: Getting dictionary of system information succeeded
{'aaa': {'fail_through': False,
                 'login_lockout_time': 300,
                 'radius_auth': 'pap',
                 'radius_retries': 1,
                 'radius_timeout': 5,
.
.
.
.
.
}
INFO:root:SUCCESS: Logout succeeded
```

Congratulations! You've just run your first AOS-CX Python workflow!

1.4 Creating Workflows

To create your own workflows, follow the structure of the three provided example workflows. Generally speaking, the key elements of a workflow script are:

1. Importing the appropriate modules from `pyaoscx`.
2. Logging into the switch(es) by calling `session.login()`.
3. Calling the functions in the imported modules to execute the desired actions (the 'core steps' of the workflow).
4. Logging out of the switch(es) by calling `session.logout()`.

If you open `configure_l2_l3_vlans.py` and compare it to `print_system_info.py`, you'll notice that the main difference between the two scripts is the core steps. Inside the `try:` block, instead of getting the system information and printing it, we have calls to create a VLAN and VLAN interface, among other operations:

```
vlan.create_vlan_and_svi(999, 'VLAN999', 'vlan999', 'vlan999',
                         'For LAB 999', '10.10.10.99/24', vlan_port_desc='### SVI for␣
↪LAB999 ###',
                         **session_dict)

# Add DHCP helper IPv4 addresses for SVI
dhcp.add_dhcp_relays('vlan999', "default", ['1.1.1.1', '2.2.2.2'], **session_dict)

# Add a new entry to the Port table if it doesn't yet exist
interface.add_l2_interface('1/1/20', **session_dict)
```

(continues on next page)

(continued from previous page)

```
# Update the Interface table entry with "user-config": {"admin": "up"}
interface.enable_disable_interface('1/1/20', **session_dict)

# Set the L2 port VLAN mode as 'access'
vlan.port_set_vlan_mode('1/1/20', "access", **session_dict)

# Set the access VLAN on the port
vlan.port_set_untagged_vlan('1/1/20', 999, **session_dict)
```

Similarly, the core steps in `cleanup_l2_l3_vlans.py` unconfigure the configuration done by `configure_l2_l3_vlans.py` by essentially performing the opposite actions:

```
# Delete all DHCP relays for interface
dhcp.delete_dhcp_relays('vlan999', "default", **session_dict)

# Delete VLAN and SVI
vlan.delete_vlan_and_svi(999, 'vlan999', **session_dict)

# Initialize L2 interface
interface.initialize_interface('1/1/20', **session_dict)
```

In the example workflows, the arguments in the function calls are hardcoded in the scripts, and the switch IP address and login credentials are provided by the user at runtime. Instead, you can have the script accept these data points, along with the arguments for the other functions, from an input file.

You can see examples of how to do this in our Github repo. This repo contains more example workflows in its `workflows` folder. You'll notice that the example workflows in the Github repo read data from the YAML files in the `sampledata` folder, and then use that data for arguments in function calls, as opposed to having the data hardcoded in each workflow. Separating the data out into input files also makes it easier for the configuration parameters to be modified.

CHAPTER 2

Function Documentation

Scroll down to the “Indices and tables” section and then click on “Module Index” to see all the available documentation for all the functions in the modules.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search