Nexus 3000/9000 스위치에서 NX-SDK Python 애플리케이션 개발, 디버깅 및 배포

목차

[소개](#)

[사전 요구 사항](#)

- [요구 사항](#)
- [사용되는 구성 요소](#)

[배경 정보](#)

[NX-SDK를 사용하여 Python 애플리케이션 개발](#)

- [NX-SDK 사용](#)
- [Python 파일 만들기](#)
- [NX-SDK 구성 요소 구현](#)
- [사용자 지정 CLI 명령 생성](#)
- [pyCmdHandler 클래스](#)
- [사용자 지정 CLI 명령 구문 예](#)
- [단일 키워드](#)
- [단일 매개 변수](#)
- [선택적 키워드](#)
- [선택적 매개 변수](#)
- [단일 키워드 및 매개변수](#)
- [여러 키워드 및 매개변수](#)
- [여러 키워드 및 매개변수(선택적 키워드 포함)](#)
- [여러 키워드 및 매개 변수(선택적 매개 변수 포함)](#)

[NX-SDK를 사용하여 Python 애플리케이션 디버그](#)

[NX-SDK를 사용하여 Python 애플리케이션 배포](#)

[관련 정보](#)

소개

이 문서에서는 Nexus 3000 및 Nexus 9000 플랫폼에서 Cisco NX-OS를 사용하는 Cisco NX-SDK(Software Development Kit)를 사용하여 Python 애플리케이션을 개발하는 워크플로를 제공합니다.

사전 요구 사항

요구 사항

이 문서에 대한 특정 요건이 없습니다.

사용되는 구성 요소

이 문서의 정보는 다음 소프트웨어 및 하드웨어 버전을 기반으로 합니다.

- 이 문서에서는 NX-SDK v1.0.0 및 NX-SDK v1.5.0을 사용합니다.
- NX-SDK v1.0.0은 Nexus 9000 플랫폼에서 NX-OS 릴리스 7.0(3)I6(1)부터, Nexus 3000 플랫폼에서 NX-OS 릴리스 7.0(3)I7(1)부터 사용할 수 있습니다
- NX-SDK v1.5.0은 NX-OS 릴리스 7.0(3)I7(3)부터 Nexus 9000 플랫폼과 Nexus 3000 플랫폼 모두에서 사용할 수 있습니다.

이 문서의 정보는 특정 랩 환경의 디바이스를 토대로 작성되었습니다. 이 문서에 사용된 모든 디바이스는 초기화된(기본) 컨피그레이션으로 시작되었습니다. 현재 네트워크가 작동 중인 경우, 모든 명령어의 잠재적인 영향을 미리 숙지하시기 바랍니다.

배경 정보

Cisco NX-SDK를 사용하면 Nexus 9000 및 Nexus 3000 플랫폼의 Cisco NX-OS에서 기본적으로 실행할 수 있는 맞춤형 애플리케이션을 개발할 수 있습니다. NX-SDK는 고객이 직접 CLI 명령 및 출력을 생성하고, 특정 이벤트에 대한 응답으로 맞춤형 syslog를 생성하고, 맞춤형 텔레메트리를 스트리밍하는 등의 기능을 제공합니다.

NX-SDK에는 SWIG(Simplified Wrapper and Interface Generator)를 사용하여 다른 언어로 번역되는 C++ API가 있습니다. 따라서 고객은 원하는 언어로 NX-SDK를 활용할 수 있습니다. 이 문서에서는 Python에서 일반적인 NX-SDK 기능의 구현을 시연하고 고객이 자체 NX-SDK Python 애플리케이션을 개발하는 워크플로를 제공합니다.

NX-SDK를 사용하여 Python 애플리케이션 개발

NX-SDK 사용

NX-SDK 응용 프로그램을 실행하려면 먼저 디바이스에서 NX-SDK 기능을 활성화해야 합니다.

```
switch(config)# feature nxsdk
```

Python 파일 만들기

NX-OS Bash 셸을 사용하여 Python 파일을 만들고 편집할 수 있습니다. Bash 셸을 사용하려면 먼저 디바이스에서 Bash 셸을 활성화해야 합니다.

```
switch(config)# feature bash-shell
```

Bash 셸을 입력하고 vi 텍스트 편집기를 사용하여 Python 파일을 만들고 편집합니다.

```
switch(config)# run bash
bash-4.2$ vi /isan/bin/nxsdk-app.py
```

참고: 모범 사례는 /isan/bin/디렉토리에 Python 파일을 생성하는 것입니다. Python 파일을 실행하려면 실행 권한이 있어야 합니다. Python 파일을 /bootflash 디렉토리 또는 하위 디렉토리에 배치하지 마십시오.

참고: NX-OS를 통해 Python 파일을 만들고 편집할 필요는 없습니다. 개발자는 자신의 로컬 환경을 사용하여 애플리케이션을 생성하고, 자신이 선택한 파일 전송 프로토콜을 사용하여 완료된 파일을 디바이스로 전송할 수 있다. 그러나 개발자가 NX-OS 유틸리티를 사용하여 스크립트를 디버깅하고 문제를 해결하는 것이 더 효율적일 수 있습니다.

NX-SDK 구성 요소 구현

개발자는 Cisco DevNet NX-SDK GitHub의 customCliPyApp 템플릿에서 NX-SDK Python 응용 프로그램을 만드는 것을 권장합니다. 이 문서의 이러한 섹션에서는 이 템플릿 내에서 이름을 사용하는 필수 구성 요소를 참조합니다.

NX-SDK Python 애플리케이션에 필요한 네 가지 주요 구성 요소가 있습니다.

1. NX-SDK는 import nx_sdk_py 명령문을 통해 애플리케이션으로 가져와야 합니다.
2. NX-SDK 응용 프로그램을 시작하고 응용 프로그램과 관련된 다양한 옵션을 수정하는 함수(일반적으로 sdkThread)입니다.
3. sdkThread 함수 내에서 사용자 지정 CLI 명령 구문 정의와 사용자 지정 CLI 명령 생성.
4. NX-SDK 애플리케이션에 의해 추가된 사용자 지정 CLI 명령을 처리하는 postCliCb 메서드가 있는 pyCmdHandler라는 클래스입니다.

sdkThread 함수

sdkThread 함수는 NX-SDK 응용 프로그램을 초기화, 추가 및 시작합니다. 함수에는 매개 변수를 전달할 필요가 없습니다. 모든 Python NX-SDK 애플리케이션에는 nx_sdk_py 라이브러리의 세 가지 메서드를 호출해야 합니다.

1. nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)는 SDK 인스턴스 개체를 반환하거나 None을 반환합니다. SDK 인스턴스 개체가 반환되면 NX-SDK 애플리케이션이 NX-OS 인프라에 성공적으로 등록됩니다. None(없음)이 반환되면 이 등록 프로세스 시 오류가 발생하고

디바이스의 syslog에 Error log(오류 로그) 항목이 나타납니다. 이 방법에 대한 설명은 여기에 나와 있습니다.

참고: NX-SDK v1.5.0부터 세 번째 부울 매개 변수를 NxSdk.getSdkInst 메서드에 전달할 수 있습니다. 이 메서드는 True인 경우 고급 예외를 활성화하고 False인 경우 고급 예외를 비활성화합니다. 이 방법은 여기에 설명되어 있습니다.

---

1. sdk.startEventLoop() 메서드. 여기서 sdk는 nx_sdk_py.NxSdk.getSdkInst() 메서드에서 반환되는 SDK 인스턴스 개체입니다. 이 메서드는 NX-SDK 애플리케이션을 시작하고 NX-OS 인프라와 상호 작용할 수 있도록 합니다. 이 방법은 여기에 설명되어 있습니다.
2. nx_sdk_py.NxSdk.__swig_destroy__(sdk) 메서드에서 sdk는 앞에 설명한 nx_sdk_py.NxSdk.getSdkInst() 메서드에서 반환하는 SDK 인스턴스 개체입니다. 이 메서드는 sdkThread 함수의 끝에 배치되므로 NX-SDK 응용 프로그램을 정상적으로 종료할 수 있습니다.

일반적으로 사용되는 몇 가지 방법은 다음과 같습니다.

- sdk.getTracer(), 여기서 sdk는 nx_sdk_py.NxSdk.getSdkInst() 메서드에서 반환되는 SDK 인스턴스 개체입니다. 이 메서드는 사용자 지정 syslog를 생성하는 데 사용할 수 있는 NxTrace 개체를 반환하고, 애플리케이션의 이벤트 기록에 이벤트와 오류를 기록합니다. 사용자 지정 syslog는 디바이스의 syslog에 나타나며(show logging logfile 명령을 통해 표시됨), 애플리케이션의 이벤트 기록에 로깅된 이벤트는 show <application-name> nxsdk event-history events 또는 show <application-name> nxsdk event-history errors 명령을 통해 표시됩니다. 이 방법에 대한 설명은 여기에 나와 있습니다. 이 메서드에서 반환한 NxTrace 개체와 관련 메서드가 여기에 설명되어 있습니다. 예를 들어, show Transceiver_DOM.py nxsdk event-history events 및 show Transceiver_DOM.py nxsdk event-history errors 명령을 통해 Transceiver_DOM.py 애플리케이션의 이벤트 기록을 볼 수 있습니다.
- sdk.getCliParser(), 여기서 sdk는 nx_sdk_py.NxSdk.getSdkInst() 메서드에서 반환되는 SDK 인스턴스 개체입니다. 이 메서드는 NxCliParser 개체를 반환합니다. 이 개체는 Python을 통해 이미 존재하는 CLI 명령을 실행하고 사용자 지정 CLI 명령을 생성하는 데 사용할 수 있습니다. 이 방법에 대한 설명은 여기에 나와 있습니다. 이 메서드에서 반환한 NxCliParser 개체와 관련 메서드에 대해서는 여기에 설명되어 있습니다.
- cliP.execShowCmd("cmd", return_type), 여기서 cliP는 sdk.getCliParser() 메서드에서 반환된 NxCliParser 개체이고, cmd는 따옴표로 캡슐화된 것을 실행하려는 show 명령이며, return_type!는 출력할 데이터 형식입니다. 데이터 형식은 R_TEXT, R_JSON 또는 R_XML일 수 있습니다. 이 방법은 여기에 설명되어 있습니다.

참고: R_JSON 및 R_XML 데이터 형식은 명령에서 해당 형식의 출력을 지원하는 경우에만 작동합니다. NX-OS에서는 출력을 요청된 데이터 형식으로 파이핑하여 명령에서 특정 데이터 형식의 출력을 지원하는지 확인할 수 있습니다. piped 명령이 의미 있는 출력을 반환하면 해당 데이터 형식이 지원됩니다. 예를 들어, show mac address-table dynamic을 실행하는 경우 | NX-OS의 json이 JSON 출력을 반환하면 NX-SDK에서도 R_JSON 데이터 형식이 지원됩니다.

- cliP.execConfigCmd(cmd_filename), 여기서 cliP는 sdk.getCliParser() 메서드에서 반환된

NxCliParser 개체이며, cmd_filename은 행으로 구분된 명령이 포함된 파일의 절대 파일 경로입니다. 이 메서드는 명령 실행의 성공을 나타내는 문자열을 반환합니다. 이 문자열에 "SUCCESS"가 있으면 모든 명령이 성공적으로 실행됩니다. 그렇지 않으면 문자열에 명령이 실행되지 못한 이유를 설명하는 예외가 포함됩니다. 이 방법에 대한 설명은 여기에 나와 있습니다.

다음과 같은 몇 가지 선택적 방법이 도움이 될 수 있습니다.

- sdk.setAppDesc('description string'), 여기서 sdk는 nx_sdk_py.NxSdk.getSdkInst() 메서드에서 반환된 SDK 인스턴스 객체입니다. 이 메서드는 NX-SDK 응용 프로그램에 대한 설명을 설정합니다. 설명은 CLI의 물음표를 통해 액세스할 수 있는 NX-OS 상황에 맞는 도움말 메뉴에 표시됩니다. 이 방법에 대한 설명은 여기에 나와 있습니다. 예를 들어, Transceiver_DOM.py라는 애플리케이션과 Returns all interfaces with DOM-capable transceivers inserted의 애플리케이션 설명은 다음과 같이 NX-OS 상황에 맞는 도움말에 나타납니다.

```
N9K-C93180LC-EX# show Tra?
 track                  Tracking information
 Transceiver_DOM.py     Returns all interfaces with DOM-capable transceivers inserted
```

- sdk.getAppName(), 여기서 sdk는 nx_sdk_py.NxSdk.getSdkinst() 메서드에서 반환되는 SDK 인스턴스 개체입니다. 이 메서드는 Python 응용 프로그램의 이름을 반환합니다. 이 방법에 대한 설명은 여기에 나와 있습니다.

사용자 지정 CLI 명령 생성

NX-SDK를 사용하는 Python 응용 프로그램에서는 sdkThread 함수 내에서 사용자 지정 CLI 명령이 생성되고 정의됩니다. 두 가지 유형의 명령이 있습니다. Show 명령 및 Config 명령을 사용합니다.

1. Show 명령은 디바이스, 컨피그레이션 또는 환경에 대한 정보를 표시합니다.
2. Config 명령은 디바이스의 컨피그레이션을 변경하며, 이는 디바이스가 주변 네트워크에 반응하는 방식을 수정합니다.

이 두 가지 방법을 사용하면 각각 show 명령과 config 명령을 생성할 수 있습니다.

- cliP.newShowCmd("cmd_name", "syntax") 여기서 cliP는 sdk.getCliParser() 메서드에서 반환된 NxCliParser 개체이고, cmd_name은 사용자 지정 NX-SDK 응용 프로그램 내부의 명령에 대한 고유한 이름이며, syntax는 명령에 사용할 수 있는 키워드와 매개 변수를 설명합니다. 이 메서드는 NxCliCmd 개체를 반환합니다. 이 개체는 여기에 설명되어 있습니다. 이 방법에 대한 설명은 여기에 나와 있습니다.

참고: 이 명령은 cliP.newCliCmd("cmd_type", "cmd_name", "syntax")의 하위 클래스입니다. 여기서 cmd_type은 CONF_CMD 또는 SHOW_CMD(구성되는 명령 유형에 따라)이고, cmd_name은 사용자 지정 NX-SDK 애플리케이션 내부의 명령에 대한 고유한 이름이며, syntax는 명령에 사용할 수 있는 키워드 및 매개변수를 설명합니다. 따라서 이 명령에 대한 API 설명서는 참조에 더 유용할 수 있습니다.

- cliP.newConfigCmd("cmd_name", "syntax") 여기서 cliP는 sdk.getCliParser() 메서드에서 반환된 NxCliParser 개체이고, cmd_name은 사용자 지정 NX-SDK 응용 프로그램 내부의 명령에 대한 고유한 이름이며, syntax는 명령에 사용할 수 있는 키워드와 매개 변수를 설명합니다. 이 메서드는 NxCliCmd 개체를 반환하며, 이는 여기에 설명되어 있습니다. 이 방법은 여기에 설명되어 있습니다.

참고: 이 명령은 cliP.newCliCmd("cmd_type", "cmd_name", "syntax")의 하위 클래스입니다. 여기서 cmd_type은 CONF_CMD 또는 SHOW_CMD(구성된 명령 유형에 따라 다름)이고, cmd_name은 사용자 지정 NX-SDK 애플리케이션 내부의 명령에 대한 고유한 이름이며, syntax는 명령에 사용할 수 있는 키워드와 매개 변수를 설명합니다. 따라서 이 명령에 대한 API 설명서는 참조에 더 유용할 수 있습니다.

두 명령 유형 모두 두 가지 다른 구성 요소가 있습니다. 매개 변수 및 키워드:

1. 매개변수는 명령의 결과를 변경하는 데 사용되는 값입니다. 예를 들어 show ip route 192.168.1.0 명령에서는 route 키워드 뒤에 IP 주소를 수락하는 매개변수가 있습니다. 이는 제공된 IP 주소를 포함하는 경로만 표시하도록 지정합니다.

2. 키워드는 프레즌스만으로 명령 결과를 변경할 수 있습니다. 예를 들어 show mac address-table dynamic 명령에서 dynamic 키워드는 동적으로 학습된 MAC 주소만 표시하도록 지정합니다.

두 구성 요소는 모두 생성될 때 NX-SDK 명령의 구문에 정의됩니다. 두 구성 요소의 특정 구현을 수정하기 위해 NxCliCmd 개체에 대한 메서드가 있습니다.

- nx_cmd.updateParam("<parameter>", "help_str", type). 여기서 nx_cmd는 cliP.newShowCmd() 또는 cliP.newConfigCmd() 메서드에서 반환한 NxCliCmd 개체이고, <parameter>는 꺾쇠 괄호(<>)로 묶어 수정할 수 있는 명령 매개 변수의 이름이며, help_str은 사용자 지정 명령의 help-string을 설정하고 CLI에서 물음표를 통해 액세스할 수 있는 NX-OS 상황에 맞는 도움말 메뉴에 표시되며, type!는 매개 변수의 유형입니다. 이 방법에 대한 추가 선택적 매개 변수를 사용할 수 있으며 여기에 설명되어 있습니다. type 인수에 지정할 수 있는 매개 변수의 올바른 형식은 다음과 같습니다.

    - P_INTEGER - 임의의 정수를 지정합니다.
    - P_STRING - 임의의 문자열을 지정합니다.
    - P_INTERFACE - 모든 네트워크 인터페이스를 지정합니다.
    - P_IP_ADDR - 모든 IP 주소를 지정합니다.
    - P_MAC_ADDR - 모든 MAC 주소를 지정합니다.
    - P_VRF - 모든 VRF(Virtual Routing and Forwarding) 인스턴스를 지정합니다.
- nx_cmd.updateKeyword("keyword", "help_str", is_key). 여기서 nx_cmd는 cliP.newShowCmd() 또는 cliP.newConfigCmd() 메서드에서 반환한 NxCliCmd 개체입니다. keyword는 수정할 command 키워드의 이름이고, help_str은 사용자 지정 명령의 help-string을 설정하며 CLI에서 물음표를 통해 액세스되는 NX-OS 상황에 맞는 도움말 메뉴에 표시되며, is_key는 False로 기본 설정되는 선택적 부울 값입니다. is_key가 True이면 이 키워드를 사용하여 명령에 의해 생성된 고유한 컨피그레이션이 명령에 의해 생성된 다른 고유한 컨피그레이션을 덮어쓰지 않습니다. is_key가 False이면 이 키워드를 사용하여 명령에 의해 생성된 컨피

그레이션이 명령에 의해 생성된 다른 컨피그레이션을 덮어씁니다. 이 방법은 여기에 설명되어 있습니다.

일반적으로 사용되는 명령 구성 요소의 코드 예를 보려면 이 문서의 Custom CLI Command Examples 섹션을 참조하십시오.

사용자 지정 CLI 명령을 만든 후에는 이 문서의 뒷부분에 설명된 pyCmdHandler 클래스의 개체를 만들고 NxCliParser 개체에 대한 CLI 콜백 처리기 개체로 설정해야 합니다. 이는 다음과 같이 입증되었습니다.

```
cmd_handler = pyCmdHandler()
cliP.setCmdHandler(cmd_handler)
```

그런 다음 맞춤형 CLI 명령이 사용자에게 표시되도록 NxCliParser 개체를 NX-OS CLI 파서 트리에 추가해야 합니다. 이 작업은 cliP.addToParseTree() 명령을 사용하여 수행됩니다. 여기서 cliP는 sdk.getCliParser() 메서드에서 반환된 NxCliParser 개체입니다.

sdkThread 함수 예

다음은 앞에서 설명한 함수를 사용하는 일반적인 sdkThread 함수의 예입니다. 이 함수는 일반 사용자 지정 NX-SDK Python 응용 프로그램 내의 여러 함수 중에서 스크립트 실행 시 인스턴스화되는 전역 변수를 사용합니다.

```
cliP = ""
sdk = ""
event_hdlr = ""
tmsg = ""

def sdkThread():
    global cliP, sdk, event_hdlr, tmsg

    sdk = nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)
    if not sdk:
       return

    sdk.setAppDesc("Returns all interfaces with DOM-capable transceivers inserted")

    tmsg = sdk.getTracer()
    tmsg.event("[{}] Started service".format(sdk.getAppName()))

    cliP = sdk.getCliParser()

    nxcmd = cliP.newShowCmd("show_port_bw_util_cmd", "port bw utilization [<port>]")
    nxcmd.updateKeyword("port", "Port Information")
    nxcmd.updateKeyword("bw", "Port Bandwidth Information")
    nxcmd.updateKeyword("utilization", "Port BW utilization in (%)")
    nxcmd.updateParam("<port>", "Optional Filter Port Ex) Ethernet1/1", nx_sdk_py.P_INTERFACE)

    nxcmd1 = cliP.newConfigCmd("port_bw_threshold_cmd", "port bw threshold <threshold>")
    nxcmd1.updateKeyword("threshold", "Port BW Threshold in (%)")
```

```
    int_attr = nx_sdk_py.cli_param_type_integer_attr()
    int_attr.min_val = 1;
    int_attr.max_val = 100;
    nxcmd1.updateParam("<threshold>", "Threshold Limit. Default 50%", nx_sdk_py.P_INTEGER, int_attr, le

    mycmd = pyCmdHandler()
    cliP.setCmdHandler(mycmd)

    cliP.addToParseTree()

    sdk.startEventLoop()

    # If sdk.stopEventLoop() is called or application is removed from VSH...
    tmsg.event("Service Quitting...!")

    nx_sdk_py.NxSdk.__swig_destroy__(sdk)
```

pyCmdHandler 클래스

pyCmdHandler 클래스는 nx_sdk_py 라이브러리 내의 NxCmdHandler 클래스에서 상속됩니다. pyCmdHandler 클래스 내에 정의된 postCliCb(self, clicmd) 메서드는 NX-SDK 애플리케이션에서 시작되는 CLI 명령이 있을 때마다 호출됩니다. 따라서 postCliCb(self, clicmd) 메서드는 sdkThread 함수에 정의된 사용자 지정 CLI 명령이 디바이스에서 작동하는 방식을 정의하는 곳입니다.

postCliCb(self, clicmd) 함수는 부울 값을 반환합니다. True가 반환되면 명령이 성공적으로 실행된 것으로 간주됩니다. 명령이 어떤 이유로든 성공적으로 실행되지 않은 경우 False를 반환해야 합니다.

clicmd 매개 변수는 sdkThread 함수에서 만들 때 명령에 대해 정의된 고유한 이름을 사용합니다. 예를 들어, show_xcvr_dom의 고유 이름을 사용하여 새 show 명령을 만들 경우 clicmd 인수의 이름에 show_xcvr_dom이 포함되어 있는지 확인한 후 postCliCb(self, clicmd) 함수에서 동일한 이름으로 이 명령을 참조하는 것이 좋습니다. 이 데모는 여기에서 확인할 수 있습니다.

```
def sdkThread():
    <snip>
    sh_xcvr_dom = cliP.newShowCmd("show_xcvr_dom", "dom")
    sh_xcvr_dom.updateKeyword("dom", "Show all interfaces with transceivers that are DOM-capable")
    </snip>

class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        if "show_xcvr_dom" in clicmd.getCmdName():
            get_dom_capable_interfaces()
```

매개 변수를 사용하는 명령이 생성되면 postCliCb(self, clicmd) 함수의 특정 지점에서 해당 매개 변수를 사용해야 할 가능성이 높습니다. 이 작업은 clicmd.getParamValue("<parameter>") 메서드로 수행할 수 있습니다. <parameter>는 각도 대괄호(<>)로 묶인 값을 가져오려는 명령 매개 변수의 이

름입니다. 이 방법에 대한 설명은 여기에 나와 있습니다. 그러나 이 함수에서 반환되는 값은 필요한 형식으로 변환해야 합니다. 이 작업은 다음 방법으로 수행할 수 있습니다.

- nx_sdk_py.void_to_int는 값을 정수 유형으로 변환합니다.
- nx_sdk_py.void_to_string 값을 문자열 유형으로 변환합니다.

postCliCb(self, clicmd) 함수(또는 후속 함수)도 일반적으로 show 명령 출력이 콘솔에 인쇄되는 곳입니다. 이 작업은 clicmd.printConsole() 메서드를 사용하여 수행됩니다.

참고: 응용 프로그램에 오류가 발생하거나 처리되지 않은 예외가 발생하거나 갑자기 종료되면 clicmd.printConsole() 함수의 출력이 전혀 표시되지 않습니다. 따라서 Python 응용 프로그램을 디버깅할 때 모범 사례는 sdk.getTracer() 메서드에서 반환된 NxTrace 개체를 사용하여 디버그 메시지를 syslog에 기록하거나 print 문을 사용하여 Bash 셸의 /isan/bin/python 이진을 통해 응용 프로그램을 실행하는 것입니다.

---

pyCmdHandler 클래스 예

다음 코드는 위에서 설명한 pyCmdHandler 클래스의 예로 사용됩니다. 이 코드는 여기에서 사용 가능한 ip-movement NX-SDK 응용 프로그램의 ip_move.py 파일에서 가져온 것입니다. 이 애플리케이션의 목적은 Nexus 디바이스의 인터페이스 전체에서 사용자 정의 IP 주소의 이동을 추적하는 것입니다. 이를 위해 코드는 디바이스의 ARP 캐시 내에서 <ip> 매개변수를 통해 입력된 IP 주소의 MAC 주소를 찾은 다음 디바이스의 MAC 주소 테이블을 사용하여 MAC 주소가 상주하는 VLAN을 확인합니다. 이 MAC 및 VLAN을 사용하여 show system internal l2fm l2dbg macdb address <mac> vlan <vlan> 명령은 이 조합이 최근에 연결된 SNMP 인터페이스 인덱스 목록을 표시합니다. 그런 다음 코드에서는 show interface snmp-ifindex 명령을 사용하여 최근 SNMP 인터페이스 인덱스를 사람이 인식할 수 있는 인터페이스 이름으로 변환합니다.

```
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        global cli_parser

        if "show_ip_movement" in clicmd.getCmdName():
            target_ip = nx_sdk_py.void_to_string(clicmd.getParamValue("<ip>"))

            target_mac = get_mac_from_arp(cli_parser, clicmd, target_ip)
            mac_vlan = ""
            if target_mac:
                mac_vlan = get_vlan_from_cam(cli_parser, clicmd, target_mac)
                if mac_vlan:
                    find_mac_movement(cli_parser, clicmd, target_mac, mac_vlan)
                else:
                    print("No entires in MAC address table")
                    clicmd.printConsole("No entries in MAC address table for {}".format(target_mac))
            else:
                clicmd.printConsole("No entries in ARP table for {}".format(target_ip))
        return True

def get_mac_from_arp(cli_parser, clicmd, target_ip):
    exec_cmd = "show ip arp {}".format(target_ip)
    arp_cmd = cli_parser.execShowCmd(exec_cmd, nx_sdk_py.R_JSON)
    if arp_cmd:
        try:
```

```
            arp_json = json.loads(arp_cmd)
        except ValueError as exc:
            return None
        count = int(arp_json["TABLE_vrf"]["ROW_vrf"]["cnt-total"])
        if count:
            intf = arp_json["TABLE_vrf"]["ROW_vrf"]["TABLE_adj"]["ROW_adj"]
            if intf.get("ip-addr-out") == target_ip:
                target_mac = intf["mac"]
                clicmd.printConsole("{} is currently present in ARP table, MAC address {}\n".format(tar
                return target_mac
            else:
                return None
        else:
            return None
    else:
        return None

def get_vlan_from_cam(cli_parser, clicmd, target_mac):
    exec_cmd = "show mac address-table address {}".format(target_mac)
    mac_cmd = cli_parser.execShowCmd(exec_cmd, nx_sdk_py.R_JSON)
    if mac_cmd:
        try:
            cam_json = json.loads(mac_cmd)
        except ValueError as exc:
            return None
        mac_entry = cam_json["TABLE_mac_address"]["ROW_mac_address"]
        if mac_entry:
            if mac_entry["disp_mac_addr"] == target_mac:
                egress_intf = mac_entry["disp_port"]
                mac_vlan = mac_entry["disp_vlan"]
                clicmd.printConsole("{} is currently present in MAC address table on interface {}, VLAN
                return mac_vlan
            else:
                return None
        else:
            return None
    else:
        return None

def find_mac_movement(cli_parser, clicmd, target_mac, mac_vlan):
    exec_cmd = "show system internal l2fm l2dbg macdb address {} vlan {}".format(target_mac, mac_vlan)
    l2fm_cmd = cli_parser.execShowCmd(exec_cmd)
    if l2fm_cmd:
        event_re = re.compile(r"^\s+(\w{3}) (\w{3}) (\d+) (\d{2}):(\d{2}):(\d{2}) (\d{4}) (0x\S{8}) (\d
        unique_interfaces = []
        l2fm_events = l2fm_cmd.splitlines()
        for line in l2fm_events:
            res = re.search(event_re, line)
            if res:
                day_name = res.group(1)
                month = res.group(2)
                day = res.group(3)
                hour = res.group(4)
                minute = res.group(5)
                second = res.group(6)
                year = res.group(7)
                if_index = res.group(8)
                db = res.group(9)
                event = res.group(10)
                src=res.group(11)
                slot = res.group(12)
                fe = res.group(13)
```

```
                if "MAC_NOTIF_AM_MOVE" in event:
                    timestamp = "{} {} {} {}:{}:{} {}".format(day_name, month, day, hour, minute, secon
                    intf_dict = {"if_index": if_index, "timestamp": timestamp}
                    unique_interfaces.append(intf_dict)
        if not unique_interfaces:
            clicmd.printConsole("No entries for {} in L2FM L2DBG\n".format(target_mac))
        if len(unique_interfaces) == 1:
            clicmd.printConsole("{} has not been moving between interfaces\n".format(target_mac))
        if len(unique_interfaces) > 1:
                clicmd.printConsole("{} has been moving between the following interfaces, from most rec
                unique_interfaces = get_snmp_intf_index(unique_interfaces)
                clicmd.printConsole("\t{} - {} (Current interface)\n".format(unique_interfaces[-1]["tim
                for intf in unique_interfaces[-2::-1]:
                    clicmd.printConsole("\t{} - {}\n".format(intf["timestamp"], intf["intf_name"]))

def get_snmp_intf_index(if_index_dict_list):
    global cli_parser

    snmp_ifindex = cli_parser.execShowCmd("show interface snmp-ifindex", nx_sdk_py.R_JSON)
    snmp_ifindex_json = json.loads(snmp_ifindex)
    snmp_ifindex_list = snmp_ifindex_json["TABLE_interface"]["ROW_interface"]
    for index_dict in if_index_dict_list:
        index = index_dict["if_index"]
        for ifindex_json in snmp_ifindex_list:
            if index == ifindex_json["snmp-ifindex"]:
                index_dict["intf_name"] = ifindex_json["interface"]
    return if_index_dict_list
```

사용자 지정 CLI 명령 구문 예

이 섹션에서는 cliP.newShowCmd() 또는 cliP.newConfigCmd() 메서드를 사용하여 사용자 지정 CLI 명령을 만들 때 사용되는 구문 매개 변수의 몇 가지 예를 보여 줍니다. 여기서 cliP는 sdk.getCliParser() 메서드에서 반환되는 NxCliParser 개체입니다.

참고: 여는 괄호와 닫는 괄호("(" 및 ")")가 있는 구문에 대한 지원은 NX-OS 릴리스 7.0(3)I7(3)에 포함된 NX-SDK v1.5.0에 도입되었습니다. 사용자가 여는 괄호와 닫는 괄호를 사용하는 구문이 포함된 지정된 예를 따를 때 NX-SDK v1.5.0을 사용하는 것으로 가정합니다.

단일 키워드

이 show 명령은 단일 키워드 mac을 사용하여 이 디바이스에서 잘못 프로그래밍된 모든 MAC 주소를 Show라는 도우미 문자열을 키워드에 추가합니다.

```
nx_cmd = cliP.newShowCmd("show_misprogrammed", "mac")
nx_cmd.updateKeyword("mac", "Shows all misprogrammed MAC addresses on this device")
```

단일 매개 변수

이 show 명령은 단일 매개 변수 <mac>을 사용합니다. mac이라는 단어 주위에 괄호를 묶으면 이것이 매개변수임을 나타냅니다. 잘못된 프로그래밍을 확인하기 위한 MAC 주소의 도우미 문자열이 매개 변수에 추가됩니다. nx_sdk_py.P_MAC_ADDR 매개 변수는 nx_cmd.updateParam() 메서드의 매개 변수 유형을 MAC 주소로 정의하는 데 사용됩니다. 이렇게 하면 문자열, 정수 또는 IP 주소와 같은 다른 유형의 최종 사용자 입력이 방지됩니다.

```
nx_cmd = cliP.newShowCmd("show_misprogrammed_mac", "<mac>")
nx_cmd.updateParam("<mac>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
```

선택적 키워드

이 show 명령은 선택적으로 단일 키워드 [mac]를 사용할 수 있습니다. mac이라는 단어 주위에 괄호를 묶으면 이 키워드는 선택 사항임을 나타냅니다. 이 디바이스에서 잘못 프로그래밍된 모든 MAC 주소를 표시하는 의 도우미 문자열이 키워드에 추가됩니다.

```
nx_cmd = cliP.newShowCmd(  "show_misprogrammed_mac"  ,   "[mac]"  )
nx_cmd.updateKeyword(  "mac"  ,   "Shows all misprogrammed MAC addresses on this device"  )
```

선택적 매개 변수

이 show 명령은 선택적으로 단일 매개 변수 [<mac>]를 사용할 수 있습니다. < mac > 단어를 묶는 괄호는 이 매개 변수가 선택 사항임을 나타냅니다. mac이라는 단어 주위에 괄호를 묶으면 이것이 매개변수임을 나타냅니다. 잘못된 프로그래밍을 확인하기 위한 MAC 주소의 도우미 문자열이 매개 변수에 추가됩니다. nx_sdk_py.P_MAC_ADDR 매개 변수는 nx_cmd.updateParam() 메서드의 매개 변수 유형을 MAC 주소로 정의하는 데 사용됩니다. 이렇게 하면 문자열, 정수 또는 IP 주소와 같은 다른 유형의 최종 사용자 입력이 방지됩니다.

```
nx_cmd = cliP.newShowCmd("show_misprogrammed_mac", "[<mac>]")
nx_cmd.updateParam("<mac>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
```

단일 키워드 및 매개변수

이 show 명령은 단일 키워드 mac 바로 뒤에 <mac-address> 매개변수를 사용합니다. mac-address라는 단어를 묶는 꺾쇠괄호는 이것이 매개변수임을 나타냅니다. Check MAC address for misprogramming의 도우미 문자열이 키워드에 추가됩니다. 잘못된 프로그래밍을 확인하기 위한 MAC 주소의 도우미 문자열이 매개 변수에 추가됩니다. nx_sdk_py.P_MAC_ADDR 매개 변수는 nx_cmd.updateParam() 메서드의 형식을 MAC 주소로 정의하기 위해 사용되는데, 이는 문자열, 정수 또는 IP 주소와 같은 다른 형식의 최종 사용자 입력을 방지합니다.

```
nx_cmd = cliP.newShowCmd("show_misprogrammed", "mac <mac-address>")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
```

여러 키워드 및 매개변수

이 show 명령은 두 개의 키워드 중 하나를 선택할 수 있으며, 두 키워드 모두 뒤에 두 개의 서로 다른 매개 변수가 있습니다. 첫 번째 키워드 mac은 <mac-address>의 파라미터를 갖고, 두 번째 키워드 ip는 <ip-address>의 파라미터를 갖는다. mac-address와 ip-address라는 단어를 괄호로 묶으면 매개 변수임을 나타냅니다. Check MAC address for misprogramming의 도우미 문자열이 mac 키워드에 추가됩니다. 잘못된 프로그래밍을 확인하기 위한 MAC 주소의 도우미 문자열이 <mac-address> 매개 변수에 추가됩니다. nx_sdk_py.P_MAC_ADDR 매개 변수는 nx_cmd.updateParam() 메서드의 형식을 MAC 주소로 정의하는 데 사용됩니다. 이렇게 하면 문자열, 정수 또는 IP 주소와 같은 다른 형식의 최종 사용자 입력이 방지됩니다. Check IP address for misprogramming의 도우미 문자열이 ip 키워드에 추가됩니다. 잘못된 프로그래밍을 확인할 IP 주소의 도우미 문자열이 <ip-address> 매개 변수에 추가됩니다. nx_sdk_py.P_IP_ADDR 매개 변수는 nx_cmd.updateParam() 메서드의 형식을 IP 주소로 정의하는 데 사용됩니다. 이렇게 하면 문자열, 정수 또는 IP 주소와 같은 다른 형식의 최종 사용자 입력이 방지됩니다.

```
nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>)")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming", nx_sdk_py.P_IP_ADDR)
```

여러 키워드 및 매개변수(선택적 키워드 포함)

이 show 명령은 두 개의 키워드 중 하나를 선택할 수 있으며, 두 키워드 모두 뒤에 두 개의 서로 다른 매개 변수가 있습니다. 제1 키워드(mac)는 <mac-address>의 파라미터를 가지고, 제2 키워드(ip)는 <ip-address>의 파라미터를 가진다. mac-address와 ip-address라는 단어를 괄호로 묶으면 매개 변수임을 나타냅니다. Check MAC address for misprogramming의 도우미 문자열이 mac 키워드에 추가됩니다. 잘못된 프로그래밍을 확인하기 위한 MAC 주소의 도우미 문자열이 <mac-address> 매개 변수에 추가됩니다. nx_sdk_py.P_MAC_ADDR 매개 변수는 nx_cmd.updateParam() 메서드의 형식을 MAC 주소로 정의하는 데 사용됩니다. 이렇게 하면 문자열, 정수 또는 IP 주소와 같은 다른 형식의 최종 사용자 입력이 방지됩니다. Check IP address for misprogramming의 도우미 문자열이 ip 키워드에 추가됩니다. 잘못된 프로그래밍을 확인할 IP 주소의 도우미 문자열이 <ip-address> 매개 변수에 추가됩니다. nx_sdk_py.P_IP_ADDR 매개 변수는 nx_cmd.updateParam() 메서드의 형식을 IP 주소로 정의하는 데 사용됩니다. 이렇게 하면 문자열, 정수 또는 IP 주소와 같은 다른 형식의 최종 사용자 입력이 방지됩니다. 이 show 명령은 선택적으로 [clear] 키워드를 사용할 수 있습니다. 헬퍼 문자열 잘못 프로그래밍된 것으로 탐지된 주소를 지웁니다. 이 선택적 키워드에 추가됩니다.

```
nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>) [clear]")
```

```
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming", nx_sdk_py.P_IP_ADDR)
nx_cmd.updateKeyword("clear", "Clears addresses detected to be misprogrammed")
```

여러 키워드 및 매개 변수(선택적 매개 변수 포함)

이 show 명령은 두 개의 키워드 중 하나를 선택할 수 있으며, 두 키워드 모두 뒤에 두 개의 서로 다른 매개 변수가 있습니다. 제1 키워드(mac)는 <mac-address>의 파라미터를 가지고, 제2 키워드(ip)는 <ip-address>의 파라미터를 가진다. mac-address와 ip-address라는 단어를 괄호로 묶으면 매개 변수임을 나타냅니다. Check MAC address for misprogramming의 도우미 문자열이 mac 키워드에 추가됩니다. 잘못된 프로그래밍을 확인하기 위한 MAC 주소의 도우미 문자열이 <mac-address> 매개 변수에 추가됩니다. nx_sdk_py.P_MAC_ADDR 매개 변수는 nx_cmd.updateParam() 메서드의 형식을 MAC 주소로 정의하는 데 사용됩니다. 이렇게 하면 문자열, 정수 또는 IP 주소와 같은 다른 형식의 최종 사용자 입력이 방지됩니다. Check IP address for misprogramming의 도우미 문자열이 ip 키워드에 추가됩니다. 잘못된 프로그래밍을 확인할 IP 주소의 도우미 문자열이 <ip-address> 매개 변수에 추가됩니다. nx_sdk_py.P_IP_ADDR 매개 변수는 nx_cmd.updateParam() 메서드의 형식을 IP 주소로 정의하는 데 사용됩니다. 이렇게 하면 문자열, 정수 또는 IP 주소와 같은 다른 형식의 최종 사용자 입력이 방지됩니다. 이 show 명령은 매개 변수 [<module>]를 선택적으로 사용할 수 있습니다. 헬퍼 문자열 지정된 모듈의 clear addresses만 이 선택적 매개 변수에 추가됩니다.

```
nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>) [<module>]")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming", nx_sdk_py.P_IP_ADDR)
nx_cmd.updateParam("<module>", "Clears addresses detected to be misprogrammed", nx_sdk_py.P_INTEGER)
```

NX-SDK를 사용하여 Python 애플리케이션 디버그

NX-SDK Python 응용 프로그램을 만든 후에는 종종 디버깅해야 합니다. NX-SDK는 코드에 구문 오류가 있을 경우 알려주지만 Python NX-SDK 라이브러리는 SWIG를 사용하여 C++ 라이브러리를 Python 라이브러리로 변환하므로 코드 실행 시 발생한 모든 예외는 다음과 유사한 애플리케이션 코어 덤프를 생성합니다.

```
terminate called after throwing an instance of 'Swig::DirectorMethodException'
  what(): SWIG director method error. Error detected when calling 'NxCmdHandler.postCliCb'
Aborted (core dumped)
```

이 오류 메시지의 모호한 특성으로 인해 Python 응용 프로그램을 디버깅하는 가장 좋은 방법은 sdk.getTracer() 메서드에서 반환된 NxTrace 개체를 사용하여 디버그 메시지를 syslog에 로깅하는 것입니다. 이는 다음과 같이 입증되었습니다.

```
#! /isan/bin/python

tracer = 0

def evt_thread():
    <snip>
    tracer = sdk.getTracer()
    tracer.event("[NXSDK-APP][INFO] Started service")
<snip>
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        global tracer
        tracer.event("[NXSDK-APP][DEBUG] Received command: {}".format(clicmd))
        if "show_test_command" in clicmd.getCmdName():
            tracer.event("[NXSDK-APP][DEBUG] `show_test_command` recognized")
```

디버그 메시지를 syslog에 로깅하는 것이 옵션이 아닌 경우, 다른 방법은 print 문을 사용하고 Bash 셸의 /isan/bin/python 이진을 통해 애플리케이션을 실행하는 것입니다. 그러나 이러한 인쇄 명령문의 출력은 이러한 방식으로 실행되는 경우에만 표시됩니다. VSH 셸을 통해 응용 프로그램을 실행하면 출력이 생성되지 않습니다. 다음은 인쇄 문장의 활용 예입니다.

```
#! /isan/bin/python

tracer = 0

def evt_thread():
    <snip>
    print("[NXSDK-APP][INFO] Started service")
<snip>
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        print("[NXSDK-APP][DEBUG] Received command: {}".format(clicmd))
        if "show_test_command" in clicmd.getCmdName():
            print("[NXSDK-APP][DEBUG] `show_test_command` recognized")
```

NX-SDK를 사용하여 Python 애플리케이션 배포

Python 애플리케이션이 Bash 셸에서 완전히 테스트되고 구축 준비가 되면 VSH를 통해 프로덕션 환경에 설치해야 합니다. 이렇게 하면 디바이스가 다시 로드되거나 시스템 전환이 듀얼 슈퍼바이저 시나리오에서 발생할 때 애플리케이션이 유지될 수 있습니다. VSH를 통해 애플리케이션을 구축하려면 NX-SDK 및 ENXOS SDK 빌드 환경을 사용하여 RPM 패키지를 생성해야 합니다. Cisco DevNet은 RPM 패키지를 손쉽게 생성할 수 있는 Docker 이미지를 제공합니다.

참고: 특정 운영 체제에 Docker를 설치하는 데 도움이 필요하면 Docker의 설치 설명서를 참조하십시오.

Docker 지원 호스트에서 docker pull dockercisco/nxsdk:<tag> 명령을 사용하여 원하는 이미지 버전을 가져옵니다. 여기서 <tag>는 선택한 이미지 버전의 태그입니다. 여기서 사용 가능한 이미지 버전 및 해당 태그를 볼 수 있습니다. 이 데모는 v1 태그를 통해 다음에서 확인할 수 있습니다.

```
docker pull dockercisco/nxsdk:v1
```

이 이미지에서 nxsdk라는 컨테이너를 시작하고 첨부합니다. 선택한 태그가 다른 경우 태그를 v1로 대체합니다.

```
docker run -it --name nxsdk dockercisco/nxsdk:v1 /bin/bash
```

최신 버전의 NX-SDK로 업데이트하고 NX-SDK 디렉토리로 이동한 다음 git에서 최신 파일을 가져옵니다.

```
cd /NX-SDK/
git pull
```

이전 버전의 NX-SDK를 사용해야 하는 경우 git clone -b v<version> https://github.com/CiscoDevNet/NX-SDK.git 명령과 함께 해당 버전 태그를 사용하여 NX-SDK 브랜치를 복제할 수 있습니다. 여기서 <version>은 필요한 NX-SDK 버전입니다. NX-SDK v1.0.0에서는 이 방법을 설명합니다.

```
cd /
rm -rf /NX-SDK
git clone -b v1.0.0 https://github.com/CiscoDevNet/NX-SDK.git
```

그런 다음 Python 애플리케이션을 Docker 컨테이너에 전송합니다. 이를 위한 몇 가지 방법이 있습니다.

- Docker 컨테이너를 종료하고(컨테이너를 중지하고 한 번 더 시작해야 함) Python 애플리케이션을 Docker 호스트로 전송한 다음 docker cp 명령을 사용하여 호스트에서 컨테이너로 애플리케이션을 복사합니다. 이 내용은 Python 애플리케이션이 Docker 호스트(/app/python_app.py)로 전송되었다는 가정 하에 여기서 확인할 수 있습니다.

```
root@2dcbe841742a:~# exit
[root@localhost ~]# docker cp /app/python_app.py nxsdk:/root/
[root@localhost ~]# docker start nxsdk
nxsdk
[root@localhost ~]# docker attach nxsdk
root@2dcbe841742a:/# ls /root/
python_app.py
```

- Python 응용 프로그램의 내용을 시스템 클립보드에 복사한 다음 vim을 사용하여 Docker 컨테이너에서 만든 파일에 붙여 넣습니다.

그런 다음 /NX-SDK/scripts/에 있는 rpm_gen.py 스크립트를 사용하여 Python 애플리케이션에서 RPM 패키지를 생성합니다. 이 스크립트에는 하나의 필수 인수와 두 개의 필수 스위치가 있습니다.

- Python 애플리케이션의 파일 이름입니다. 예를 들어, python_app.py라는 파일의 Python 애플리케이션은 python_app.py의 인수를 생성합니다. 이 파일 이름은 나중에 NX-SDK의 애플리케이션 이름으로 사용되며 NX-OS에서도 이 애플리케이션에서 생성된 명령을 참조하기 위해 사용됩니다.

✎ 참고: 파일 이름에는 .py와 같은 파일 확장자가 포함될 필요가 없습니다. 이 예에서 파일 이름이 python_app.py 대신 python_app이면 문제 없이 RPM 패키지가 생성됩니다.

- -s 스위치는 위에서 언급한 파일 이름이 있는 곳으로 연결되는 절대 파일 경로에 대한 인수를 사용합니다. 예를 들어 python_app.py가 /root/에 있으면 올바른 인수는 -s /root/입니다.
- -u 스위치는 소스 파일 이름이 실행 파일 이름과 같음을 나타냅니다.

rpm_gen.py 스크립트의 사용이 여기에 표시됩니다.

```
root@7bfd1714dd2f:~# python /NX-SDK/scripts/rpm_gen.py test_python_app -s /root/ -u
##############################################################################################
Generating rpm package...
<snip>
RPM package has been built
##############################################################################################

SPEC file: /NX-SDK/rpm/SPECS/test_python_app.spec
RPM file : /NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm
```

RPM 패키지의 파일 경로는 rpm_gen.py 스크립트 출력의 마지막 줄에 표시됩니다. 이 파일은 애플리케이션을 실행할 Nexus 디바이스로 전송할 수 있도록 Docker 컨테이너에서 호스트로 복사해야 합니다. Docker 컨테이너를 종료한 후에는 docker cp <container>:<container_filepath> <host_filepath> 명령을 사용하여 쉽게 수행할 수 있습니다. <container>는 NX-SDK Docker 컨테이너의 이름(이 경우 nxsdk)이고, <container_filepath>는 컨테이너 내의 RPM 패키지의 전체 파일 경로입니다(이 경우 /NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm), <host_filepath>는 RPM 패키지를 전송할 Docker 호스트의 전체 파일 경로입니다(이 경우, /root/)입니다. 이 명령은 여기에서 확인할 수 있습니다.

```
root@7bfd1714dd2f:/# exit
[root@localhost ~]# docker cp nxsdk:/NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm /root/
[root@localhost ~]# ls /root/
anaconda-ks.cfg             test_python_app-1.0-1.0.0.x86_64.rpm
```

원하는 파일 전송 방법을 사용하여 이 RPM 패키지를 Nexus 디바이스로 전송합니다. RPM 패키지가 디바이스에 설치되면 SMU와 마찬가지로 설치하고 활성화해야 합니다. 이는 RPM 패키지가 디바이스의 부트플래시로 전송되었다는 가정하에 다음과 같이 증명됩니다.

```
N9K-C93180LC-EX# install add bootflash:test_python_app-1.0-1.0.0.x86_64.rpm
[####################] 100%
Install operation 27 completed successfully at Tue May  8 06:40:13 2018
N9K-C93180LC-EX# install activate test_python_app-1.0-1.0.0.x86_64
[####################] 100%
Install operation 28 completed successfully at Tue May  8 06:40:20 2018
```

✎ 참고: install add 명령으로 RPM 패키지를 설치할 때 스토리지 디바이스와 패키지의 정확한 파일 이름을 포함합니다. 설치 후 RPM 패키지를 활성화할 때 스토리지 디바이스 및 파일 이름을 포함하지 마십시오. 패키지 자체의 이름을 사용하십시오. show install inactive 명령을 사용하여 패키지 이름을 확인할 수 있습니다.

RPM 패키지가 활성화되면 nxsdk service <application-name> 구성 명령을 사용하여 NX-SDK를 사용하여 애플리케이션을 시작할 수 있습니다. 여기서 <application-name>은 이전에 rpm_gen.py 스크립트가 사용되었을 때 정의된 Python 파일 이름(및 그 이후의 애플리케이션)의 이름입니다. 이는 다음과 같이 입증되었습니다.

```
N9K-C93180LC-EX# conf
Enter configuration commands, one per line. End with CNTL/Z.
N9K-C93180LC-EX(config)# nxsdk service-name test_python_app
% This could take some time. "show nxsdk internal service" to check if your App is Started & Runnning
```

show nxsdk internal service 명령으로 애플리케이션이 실행 중이고 실행을 시작했는지 확인할 수 있습니다.

```
N9K-C93180LC-EX# show nxsdk internal service

NXSDK Started/Temp unavailabe/Max services : 1/0/32
NXSDK Default App Path         : /isan/bin/nxsdk
NXSDK Supported Versions       : 1.0

Service-name              Base App        Started(PID)      Version    RPM Package
------------------------- --------------- ----------------- ---------- ------------------------
test_python_app           nxsdk_app4      VSH(23195)        1.0        test_python_app-1.0-1.0.0.x86_64
```

또한 NX-OS에서 이 애플리케이션에 의해 생성된 사용자 지정 CLI 명령에 액세스할 수 있는지 확인할 수 있습니다.

```
N9K-C93180LC-EX# show test?
  test_python_app    Nexus Sdk Application
```

관련 정보

- NX-SDK 깃허브
- Cisco Nexus 9000 Series NX-OS 프로그래밍 가이드, 릴리스 7.x
- Cisco Nexus 3000 Series NX-OS 프로그래밍 가이드, 릴리스 7.x
- Cisco Nexus 3500 Series NX-OS 프로그래밍 가이드, 릴리스 7.x
- Cisco Nexus 9000 Series 스위치를 통한 네트워크 프로그래밍 기능 및 자동화 백서
- Cisco Open NX-OS를 통한 프로그래밍 기능 및 자동화(PDF)
- 기술 지원 및 문서 – Cisco Systems

이 번역에 관하여

Cisco는 전 세계 사용자에게 다양한 언어로 지원 콘텐츠를 제공하기 위해 기계 번역 기술과 수작업 번역을 병행하여 이 문서를 번역했습니다. 아무리 품질이 높은 기계 번역이라도 전문 번역가의 번역 결과물만큼 정확하지는 않습니다. Cisco Systems, Inc.는 이 같은 번역에 대해 어떠한 책임도 지지 않으며 항상 원본 영문 문서(링크 제공됨)를 참조할 것을 권장합니다.