Cisco APIC Python API Documentation

Release 0.1

Jul 02, 2020

Contents

Contents:

CHAPTER 1

Understanding the Cisco Application Policy Infrastructure Controller

1.1 Understanding the Cisco Application Policy Infrastructure Controller

The Cisco Application Policy Infrastructure Controller (APIC) is a key component of an Application Centric Infrastructure (ACI), which delivers a distributed, scalable, multi-tenant infrastructure with external end-point connectivity controlled and grouped via application centric policies. The APIC is the key architectural component that is the unified point of automation, management, monitoring and programmability for the Application Centric Infrastructure. The APIC supports the deployment, management and monitoring of any application anywhere, with a unified operations model for physical and virtual components of the infrastructure.

The APIC programmatically automates network provisioning and control based on the application requirements and policies. It is the central control engine for the broader cloud network, simplifying management while allowing tremendous flexibility in how application networks are defined and automated.

1.1.1 ACI Policy Theory

The ACI policy model is an object-oriented model based on promise theory. Promise theory is based on scalable control of intelligent objects rather than more traditional imperative models, which can be thought of as a top-down management system. In this system, the central manager must be aware of both the configuration commands of underlying objects and the current state of those objects. Promise theory, in contrast, relies on the underlying objects to handle configuration state changes initiated by the control system itself as "desired state changes." The objects are then responsible for passing exceptions or faults back to the control system. This approach reduces the burden and complexity of the control system and allows greater scale. This system scales further by allowing the methods of underlying objects to request state changes from one another and from lower-level objects.

Within this theoretical model, ACI builds an object model for the deployment of applications, with the applications as the central focus. Traditionally, applications have been restricted by the capabilities of the network and by requirements to prevent misuse of the constructs to implement policy. Concepts such as addressing, VLAN, and security have been tied together, limiting the scale and mobility of the application. As applications are being redesigned for mobility and web scale, this traditional approach hinders rapid and consistent deployment. The ACI policy model does not dictate

anything about the structure of the underlying network. However, as dictated by promise theory, it requires some edge element, called an iLeaf, to manage connections to various devices.

1.1.2 Object Model

At the top level, the ACI object model is built on a group of one or more tenants, allowing the network infrastructure administration and data flows to be segregated. Tenants can be used for customers, business units, or groups, depending on organizational needs. For instance, an enterprise may use one tenant for the entire organization, and a cloud provider may have customers that use one or more tenants to represent their organizations. Tenants can be further divided into contexts, which directly relate to Virtual Routing and Forwarding (VRF) instances, or separate IP spaces. Each tenant can have one or more contexts, depending on the business needs of that tenant. Contexts provide a way to further separate the organizational and forwarding requirements for a given tenant. Because contexts use separate forwarding instances, IP addressing can be duplicated in separate contexts for multitenancy.

Within the context, the model provides a series of objects that define the application. These objects are endpoints (EP) and endpoint groups (EPGs) and the policies that define their relationship. Note that policies in this case are more than just a set of access control lists (ACLs) and include a collection of inbound and outbound filters, traffic quality settings, marking rules, and redirection rules. The combination of EPGs and the policies that define their interaction is an Application Network Profile in the ACI model.

1.1.3 Understanding the Management Information Tree

The Management Information Tree (MIT) consists of hierarchically organized MOs that allow you to manage the APIC. Each node in this tree is an MO and each has a unique distinguished name (DN) that identifies the MO and its place in the tree. Each MO is modeled as a Linux directory that contains all properties in an MO file and all child MOs as subdirectories.

1.1.4 Understanding Managed Objects

The APIC system configuration and state are modeled as a collection of managed objects (MOs), which are abstract representations of a physical or logical entity that contain a set of configurations and properties. For example, servers, chassis, I/O cards, and processors are physical entities represented as MOs; resource pools, user roles, service profiles, and policies are logical entities represented as MOs. Configuration of the system involves creating MOs, associating them with other MOs, and modifying their properties.

At runtime all MOs are organized in a tree structure called the Management Information Tree, providing structured and consistent access to all MOs in the system.

1.1.5 Endpoint Groups

EPGs are a collection of similar endpoints representing an application tier or set of services. They provide a logical grouping of objects that require similar policy. For example, an EPG could be the group of components that make up an application's web tier. Endpoints are defined using the network interface card (NIC), virtual NIC (vNIC), IP address, or Domain Name System (DNS) name, with extensibility to support future methods of identifying application components.

EPGs are also used to represent entities such as outside networks, network services, security devices, and network storage. EPGs are collections of one or more endpoints that provide a similar function. They are a logical grouping with a variety of use options, depending on the application deployment model in use.

1.1.6 Endpoint Group Relationships

EPGs are designed for flexibility, allowing their use to be tailored to one or more deployment models that the customer can choose. The EPGs are then used to define the elements to which policy is applied. Within the network fabric, policy is applied between EPGs, therefore defining the way that EPGs communicate with one another. This approach is designed to be extensible in the future to policy application within the EPGs.

Here are some examples of EPG use:

- EPG defined by traditional network VLANs: All endpoints connected to a given VLAN placed in an EPG
- EPG defined by Virtual Extensible LAN (VXLAN): Same as for VLANs except using VXLAN
- EPG mapped to a VMware port group
- EPG defined by IP or subnet: for example, 172.168.10.10 or 172.168.10
- EPG defined by DNS names or DNS ranges: for instance, example.foo.com or *.web.foo.com

The use of EPGs is both flexible and extensible. The model is intended to provide tools to build an application network model that maps to the actual environment's deployment model. The definition of endpoints also is extensible, providing support for future product enhancements and industry requirements. The EPG model offers a number of management advantages. It offers a single object with uniform policy to higher-level automation and orchestration tools. Tools need not operate on individual endpoints to modify policies. Additionally, it helps ensure consistency across endpoints in the same group regardless of their placement in the network.

1.1.7 Policy Enforcement

The relationship between EPGs and policies can be thought of as a matrix with one axis representing the source EPG (sEPG) and the other representing the destination EPG (dEPG.) One or more policies will be placed at the intersection of the appropriate sEPGs and dEPGs. The matrix will be sparsely populated in most cases because many EPGs have no need to communicate with one another.

Policies are divided by filters for quality of service (QoS), access control, service insertion, etc. Filters are specific rules for the policy between two EPGs. Filters consist of inbound and outbound rules: permit, deny, redirect, log, copy, and mark. Policies allow wildcard functions in the definitions. Policy enforcement typically uses a most-specific-match-first approach.

1.1.8 Application Network Profiles

An Application Network Profile is a collection of EPGs, their connections, and the policies that define those connections. Application Network Profiles are the logical representation of an application and its interdependencies in the network fabric. Application Network Profiles are designed to be modeled in a logical way that matches the way that applications are designed and deployed. The configuration and enforcement of policies and connectivity is handled by the system rather than manually by an administrator.

These general steps are required to create an Application Network Profile:

1. Create EPGs (as discussed earlier).
2. Create policies that define connectivity with these rules:
    - Permit
    - Deny
    - Log
    - Mark

- Redirect
- Copy

3. Create connection points between EPGs using policy constructs known as contracts.

1.1.9 Contracts

Contracts define inbound and outbound permit, deny, and QoS rules and policies such as redirect. Contracts allow both simple and complex definition of the way that an EPG communicates with other EPGs, depending on the requirements of the environment. Although contracts are enforced between EPGs, they are connected to EPGs using provider-consumer relationships. Essentially, one EPG provides a contract, and other EPGs consume that contract.

The provider-consumer model is useful for a number of purposes. It offers a natural way to attach a "shield" or "membrane" to an application tier that dictates the way that the tier interacts with other parts of an application. For example, a web server may offer HTTP and HTTPS, so the web server can be wrapped in a contract that allows only these services. Additionally, the contract provider-consumer model promotes security by allowing simple, consistent policy updates to a single policy object rather than to multiple links that a contract may represent. Contracts also offer simplicity by allowing policies to be defined once and reused many times.

1.1.10 Application Network Profile

The three tiers of a web application defined by EPG connectivity and the contracts constitute an Application Network Profile. Contracts also provide reusability and policy consistency for services that typically communicate with multiple EPGs.

1.1.11 Configuration Options

The Cisco Application Policy Infrastructure Controller (APIC) supports multiple configuration methods, including a GUI, a REST API, a Python API, Bash scripting, and a command-line interface.

1.1.12 Understanding Python

Python is a powerful programming language that allows you to quickly build applications to help support your network. For more information, see 'http:www.python.org <http://www.python.org>'

1.1.13 Understanding the Python API

The Python API provides a Python programming interface to the underlying REST API, allowing you to develop your own applications to control the APIC and the network fabric, enabling greater flexibility in infrastructure automation, management, monitoring and programmability.

The Python API supports Python versions 2.7 and 3.4.

1.1.14 Understanding the REST API

The APIC REST API is a programmatic interface to the APIC that uses a Representational State Transfer (REST) architecture. The API accepts and returns HTTP or HTTPS messages that contain JavaScript Object Notation (JSON) or Extensible Markup Language (XML) documents. You can use any programming language to generate the messages and the JSON or XML documents that contain the API methods or managed object (MO) descriptions.

For more information about the APIC REST API, see the *APIC REST API User Guide*.

CHAPTER 2

Installing the Cisco APIC Python SDK

2.1 Installation Requirements:

The Cisco APIC Python SDK ("cobra") comes in two installable .whl files that are part of the **cobra** namespace, they operate as one **virtual** namespace. Those installable packages are:

1. **acicobra** - This is the SDK and includes the following namespaces:
    - **cobra**
    - **cobra.mit**
    - **cobra.internal**
2. **acimodel** - This includes the Python packages that model the Cisco ACI Management Information Tree and includes the following namespaces:
    - **cobra**
    - **cobra.model**

In this document, the **acicobra** package is also referred to as **the SDK**.

Both packages are required. You can download the two .whl files from a running instance of APIC at this URL:

- http[s]://<APIC address>/cobra/_downloads/

The /cobra/_downloads directory contains the two .whl files along with the .egg files. The egg files are only for backward compatibility and users should migrate to .whl files. The actual filenames may contain extra information such as the APIC and Python versions, as shown in this example:

```
Index of cobra/_downloads

    Parent Directory
    acicobra-4.2_2j-py2.py3-none-any.whl
    acimodel-4.2_2j-py2.py3-none-any.whl
```

In this example, each .whl filename references the APIC version 4.2(2j) from which it was created and the Python version py2 and py3 with which it is compatible. The whl files are platform independent.

Download both files from APIC to a convenient directory on your host computer. We recommend placing the files in a directory with no other files.

Before installing the SDK, ensure that you have the following packages installed:

- Python 2.7 or Python3.6 - For more information, see https://www.python.org/.
- pip - For more information, see https://pypi.python.org/pypi/pip.
- virtualenv - We recommend installing the Python SDK within a virtual environment using virtualenv. A virtual environment allows isolation of the Cobra Python environment from the system Python environment or from multiple Cobra versions.For more information, see https://pypi.python.org/pypi/virtualenv.

Note: SSL support for connecting to the APIC and fabric nodes using HTTPS is present by default in the normal installation. If you intend to use the CertSession class with pyopenssl, see *Installing pyopenssl*.

Note: The model package depends on the SDK package; be sure to install the SDK package first.

2.2 Installing the SDK on Unix and Linux:

Follow these steps to install the SDK on Unix and Linux:

1. Uninstall previous SDK versions:

   ```
   pip uninstall acicobra
   ```

   If no previous versions are installed, skip this step.

2. **(Optional)Create and activate a new virtual environment in which to run the SDK.** Refer to the documentation for virtualenv or similar virtual environment tools for your operating system. If you create a virtual environment for the SDK, perform the remaining steps in the virtual environment.

3. Copy the .whl files to your development system.

4. Install the whl file using the following command:

   From a local directory (relative or absolute):

   ```
   pip install *directory/path*/acicobra
   ```

   In the following example, the .whl file is in a directory named cobra-whls that is a sub-directory of the current directory:

   ```
   $ pip install ./cobra-whls/acicobra-4.2_2j-py2.py3-none-any.whl
   ```

   Note: To install the package directly into the user-site-packages directory, use the **pip install –user** option:

   ```
   pip install --user *directory/path*/acicobra
   ```

   Note: If you intend to use the CertSession class with pyopenssl, see *Installing pyopenssl*.

2.3 Installing the SDK on Windows:

Follow these steps to install the SDK on Windows:

1. Uninstall previous SDK versions (can be skipped if previous versions have not been installed):

```
pip uninstall acicobra
```

   If no previous versions are installed, skip this step.

2. (Optional - if you want SSL support) Install OpenSSL for Windows:

   a) Install the latest Visual C++ Redistributables package from http://slproweb.com/products/Win32OpenSSL.html.

   b) Install the latest Win32 or Win64 Open SSL Light version from http://slproweb.com/products/Win32OpenSSL.html

   c) Add either C:OpenSSL-Win32bin or C:OpenSSL-Win64bin to your Windows path file.

   d) Open a command window and enter one of the following commands to add an OpenSSL path depending on which platform you have:

   - For 32-bit Windows:

```
set OPENSSL_CONF=C:\OpenSSL-Win32\bin\openssl.cfg
```

   - For 64-bit Windows

```
set OPENSSL_CONF=C:\OpenSSL-Win64\bin\openssl.cfg
```

3. Install the latest Python 2.7 version from https://www.python.org/downloads/.

4. Add the following to your Windows path:

```
;C:\Python27;C:\Python27\Scripts
```

5. Download and run https://bootstrap.pypa.io/get-pip.py to install pip and setuptools.

6. Run the following commands to install virtual environment tools:

```
pip install virtualenv
pip install virtualenv-clone
pip install virtualenvwrapper-win
```

7. Create and activate a new virtual environment.

```
mkvirtualenv acienv
```

   Note: Virtual environments using virtualenvwrapper-win are created in *%USERPROFILE%Envs* by default.

8. Upgrade pip in the virtual environment.

```
c:\users\username\Envs\acienv
python -m pip install --upgrade pip
```

9. Install the APIC Python SDK (Cobra) using the following command.

   From a local directory (relative or absolute):

```
pip install \*directory\path*\acicobra
```

   In the following example, the .whl file is in a directory named cobra-whls that is a sub-directory of the current directory:

```
> pip install cobra-whls\acicobra-4.2_2j-py2.py3-none-any.whl
```

Note: To install the package directly into the user-site-packages directory, use the **pip install –user** option.

Note: If you intend to use the CertSession class with pyopenssl, see *Installing pyopenssl*.

2.4 Installing the model package on any platform

The model package depends on the SDK package. Install the SDK package prior to installing the model package. If you uninstall the SDK package and then try to import the model package, the APIC displays an **ImportError** for the module **mit.meta**.

Installation of the model package can be accomplished via pip:

```
pip install *directory/path*/acimodel-*version*-py2.7.whl
```

In the following example, the .whl file is in a directory named cobra-whls that is a sub-directory of the current directory:

```
pip install ./cobra-whls/acimodel-4.2_2j-py2.py3-none-any.whl
```

Note: The .whl file name might be different depending on whether the file is downloaded from the APIC or from Cisco.com.

Note: If you uninstall the SDK package and then try to import the model package, the APIC displays an ImportError for the module mit.meta.

CHAPTER 3

Viewing the status of the SDK and model packages install

To view which version of the SDK and which dependancies have been installed use pip as follows:

```
pip freeze
```

Once you know the name of a package you can also use the following to show the packages dependancies:

```
pip show <packagename>
```

For example:

```
$ pip show acimodel
---
Name: acimodel
Version: 4.2_2j
Location: /local/lib/python2.7/site-packages/acimodel-4.2_2j-py2.py3-none-
↪any.whl
Requires: acicobra
```

When you install the SDK without SSL support it will depend on the following modules:

1. requests
2. future

When you install the SDK with SSL support it will depend on the following modules:

1. requests
2. future
3. pyOpenSSL

These dependancies may have their own dependancies and may require a compiler depending on your platform and method of installation.

CHAPTER 4

Uninstalling the Cisco APIC Python SDK

To uninstall the Python SDK and/or model, use pip as follows:

```
pip uninstall acicobra
pip uninstall acimodel
```

Note: If you used sudo to install the Python SDK and/or model, use **sudo pip uninstall acicobra** to uninstall the SDK and **sudo pip uninstall acimodel** to unistall the model package.

Note: Uninstalling one of the packages and not the other may leave your environment in a state where it will throw import errors when trying to import various parts of the cobra namespace. The packages should be installed together and uninstalled together.

CHAPTER 5

Installing pyopenssl

SSL support for connecting to the APIC and fabric nodes using HTTPS is present by default in the normal installation. Installing pyopenssl is necessary only if you intend to use the CertSession class with pyopenssl. Note that CertSession works with native OS calls to openssl.

Installations with SSL can require a compiler.

5.1 Installing pyopenssl

In *Installing the SDK on Unix and Linux*, substitute the following procedure for the step where the SDK .whl file is installed. If you have created a virtual environment for the SDK, enter the command in the virtual environment.

1. Upgrade pip.

```
python -m pip install --upgrade pip
```

2. Install pyopenssl with wheel.

```
pip install --use-wheel pyopenssl
```

Note: This package installs pyopenssl, cryptography, cffi, pycparser and six.

3. Install the SDK .whl file using the following command:

From a local directory (relative or absolute) you must use the –find-links option and the [ssl] option:

```
pip install \*directory\path*\acicobra
```

In the following example, the .whl file is in a directory named cobra-whls that is a sub-directory of the current directory:

```
> pip install ./cobra-whls/acicobra-4.2_2j-py2.py3-none-any.whl
```

CHAPTER 6

Getting Started with the Cisco APIC Python API

The following sections describe how to get started when developing with the APIC Python API.

6.1 Preparing for Access

A typical APIC Python API program contains the following initial setup statements, which are described in the following sections:

```
from cobra.mit.access import MoDirectory
from cobra.mit.session import LoginSession
```

6.1.1 Path Settings

If you installed the cobra sdk wheel file in the standard python site-packages, the modules are already included in the python path.

If you installed it in a different directory, add the SDK directory to your PYTHONPATH environment variable. You can alternatively use the python **sys.path.append** method to specify or update a path as shown by any of these examples:

```
import sys
sys.path.append('your_sdk_path')
```

6.2 Connecting and Authenticating

To access the APIC, you must log in with credentials from a valid user account. To make configuration changes, the account must have administrator privileges in the domain in which you will be working. Specify the APIC management IP address and account credentials in the **LoginSession** object to authenticate to the APIC as shown in this example:

```
apicUrl = 'https://192.168.10.80'
loginSession = LoginSession(apicUrl, 'admin', 'mypassword')
moDir = MoDirectory(loginSession)
moDir.login()
# Use the connected moDir queries and configuration...
moDir.logout()
```

If multiple AAA login domains are configured, you must prepend the username with "apic:*domain*\" as in this example:

```
loginSession = LoginSession(apicUrl, 'apic:CiscoDomain\\admin', 'mypassword')
```

A successful login returns a reference to a directory object that you will use for further operations. In the implementation of the management information tree (MIT), managed objects (MOs) are represented as directories.

6.3 Object Lookup

Use the **MoDirectory.lookupByDn** to look up an object within the MIT object tree by its distinguished name (DN). This example looks for an object called 'uni':

```
uniMo = moDir.lookupByDn('uni')
```

A successful lookup operation returns a reference to the object that has the specified DN.

You can also look up an object by class. This example returns a list of all objects of the class 'polUni':

```
uniMo = moDir.lookupByClass('polUni')
```

You can add a filter to a lookup to find specific objects. This example returns an object of class 'fvTenant' whose name is 'Tenant1':

```
tenant1Mo = moDir.lookupByClass("fvTenant", propFilter='and(eq(fvTenant.name, "Tenant1
↪"))')
```

You can also look up an object using the dnquery class or the class query class. For more information, see the Request module.

6.4 Object Creation

The following example shows the creation of a tenant object:

```
from cobra.model.fv import Tenant
fvTenantMo = Tenant(uniMo, 'Tenant1')
```

In this example, the command creates an object of the fv.Tenant class and returns a reference to the object. The tenant object is named 'Tenant1' and is created under an existing 'uni' object referenced by 'uniMo.' An object can be created only under an object of a parent class to the class of the object being created. See the *Cisco APIC Management Information Model Reference* to determine the legal parent classes of an object you want to create.

6.5 Querying Objects

You can use the **MoDirectory.query** function to query an object within the APIC configuration, such as an application, tenant, or port. For example:

```
from cobra.mit.request import DnQuery
dnQuery = DnQuery(fvTenantMo.dn)
dnQuery.queryTarget = 'children'
childMos = moDir.query(dnQuery)
```

6.6 Committing a Configuration

Use the **MoDirectory.commit** function to save a new configuration to the mit:

```
from cobra.mit.request import ConfigRequest
cfgRequest = ConfigRequest()
cfgRequest.addMo(fvTenantMo)
moDir.commit(cfgRequest)
```

CHAPTER 7

API Reference

The Application Policy Infrastructure Controller (APIC) Python API allows you to create your own applications for manipulating the APIC configuration.

The available packages are as follows:

7.1 Naming Module

The APIC system configuration and state are modeled as a collection of managed objects (MOs), which are abstract representations of a physical or logical entity that contain a set of configurations and properties. For example, servers, chassis, I/O cards, and processors are physical entities that are represented as MOs; resource pools, user roles, service profiles, and policies are logical entities represented as MOs.

At runtime, all MOs are organized in a tree structure, which is called the Management Information Tree (MIT). This tree provides structured and consistent access to all MOs in the system. Each MO is identified by its relative name (RN) and distinguished name (DN). You can manage MO naming by using the naming module of the Python API.

You can use the naming module to create and parse object names, as well as access a variety of information about the object, including the relative name, parent or ancestor name, naming values, meta class, or MO class. You can also perform operations on an MO such as appending an Rn to a Dn or cloning an MO.

7.1.1 Relative Name (RN)

A relative name (RN) identifies an object from its siblings within the context of the parent MO. An Rn is a list of prefixes and properties that uniquely identify the object from its siblings.

For example, the Rn for an MO of type aaaUser is user-john. user- is the naming prefix and john is the *name* value.

You can use an RN class to convert between an MO's RN and constituent naming values.

The string form of an RN is {prefix}{val1}{prefix2}{Val2} (...)

Note: The naming value is enclosed in brackets ([]) if the meta object specifies that properties be delimited.

class cobra.mit.naming.**Rn**(*classMeta*, **namingVals*)

The Rn class is the relative name (Rn) of the managed object (MO). You can use Rn to convert between Rn of an MO its constituent naming values. The string form of Rn is {prefix}{val1}{prefix2}{Val2} (...) Note: The naming value is enclosed in brackets ([]) if the meta object specifies that properties be delimited.

__eq__(*other*)

Implement ==.

__ge__(*other*)

Implement >=.

__gt__(*other*)

Implement >.

__init__(*classMeta*, **namingVals*)

Relative Name (Rn) of the Mo from class meta and list of naming values

Parameters

- **classMeta** (cobra.mit.meta.ClassMeta) – class meta of the mo class
- **namingVals** (*list*) – list of naming values

__le__(*other*)

Implement <=.

__lt__(*other*)

Implement <.

__ne__(*other*)

Implement !=.

classmethod fromString(*classMeta*, *rnStr*)

Create a relative name object from the string form given the class meta

Parameters

- **classMeta** (cobra.mit.meta.ClassMeta) – class meta of the mo class
- **rnStr** (*str*) – string form of rn

Returns Rn object

Return type *cobra.mit.naming.Rn*

meta

class meta of the mo class for this Rn

Returns class meta of the mo for this Rn

Return type *cobra.mit.meta.ClassMeta*

moClass

Mo class for this Rn

Returns Mo class for this Rn

Return type *cobra.mit.mo.Mo*

namingVals

Iterator of naming values for this rn

Returns iterator of the naming values for this rn

Return type iterator

7.1.2 Distinguished Name (DN)

A distinguished name (DN) uniquely identifies a managed object (MO). A DN is an ordered list of relative names, such as the following:

dn = rn1/rn2/rn3/....

In the next example, the DN provides a fully qualified path for user-john from the top of the MIT to the MO.

dn = "uni/userext/user-john"

This DN consists of these relative names:

Relative Name	Class	Description
uni	polUni	Policy universe
userext	aaaUserEp	User endpoint
user-john	aaaUser	Local user account

Note: When using the API to filter by distinguished name (DN), we recommend that you use the full DN rather than a partial DN.

class cobra.mit.naming.**Dn**(*rns=None*)

The distinguished name (Dn) uniquely identifies a managed object (MO). A Dn is an ordered list of relative names, such as:

dn = rn1/rn2/rn3/....

In this example, the Dn provides a fully qualified path for **user-john** from the top of the Mit to the Mo.

dn = "uni/userext/user-john"

__eq__(*other*)

Implement ==.

__ge__(*other*)

Implement >=.

__gt__(*other*)

Implement >.

__init__(*rns=None*)

Create a Dn from list of Rn objects.

Parameters **rns** (*list*) – list of Rns

__le__(*other*)

Implement <=.

__lt__(*other*)

Implement <.

__ne__(*other*)

Implement !=.

appendRn(*rn*)
Appends an Rn to this Dn, changes the target Mo

clone()
Return a new copy of this Dn

Returns copy of this Dn

Return type *cobra.mit.naming.Dn*

classmethod findCommonParent(*dns*)
Find the common parent for the given set of dn objects.

Parameters **dns** (*list*) – list of Dn objects

Returns Dn object of the common parent if any, else Dn for topRoot

Return type *cobra.mit.naming.Dn*

classmethod fromString(*dnStr*)
Create a Dn from the string form of Dn. This method parses the dn string into its constituent Rn strings and creates the Rn objects.

Parameters **dnStr** (*str*) – string form of Dn

Returns Dn object

Return type *cobra.mit.naming.Dn*

getAncestor(*level*)
Returns the ancestor Dn based on the number of levels

Parameters **level** (*int*) – number of levels

Returns Dn object of the ancestor as specified by the level param

Return type *cobra.mit.naming.Dn*

getParent()

Returns the parent Dn, same as:: self.getAncetor(1)

Returns Dn object of the immediate parent

Return type *cobra.mit.naming.Dn*

isAncestorOf(*descendantDn*)
Return True if this Dn is an ancestor of the other Dn

Parameters **descendantDn** (cobra.mit.naming.Dn) – Dn being compared for descendants

Returns True if this Dn is an ancestor of the other Dn else False

Return type boolean

isDescendantOf(*ancestorDn*)
Return True if this Dn is a descendant of the other Dn

Parameters **ancestorDn** (cobra.mit.naming.Dn) – Dn being compared for ancestory

Returns True if this Dn is a descendant of the other Dn else False

Return type boolean

meta
class meta of the mo class for this Dn

Returns class meta of the mo for this Dn

Return type *cobra.mit.meta.ClassMeta*

moClass
Mo class for this Dn

Returns Mo class for this Dn

Return type *cobra.mit.mo.Mo*

rn(*index=None*)
Returns the Rn object at the specified index. If index is None, then the Rn of the target Mo is returned

Parameters **index** (*int*) – index of the Rn object, this must be betwee 0 and the length of the Dn

Returns Rn object at the specified index

Return type *cobra.mit.naming.Rn*

rns
Iterator for all the rns from topRoot to the target Mo

Returns iterator of Rns in this Dn

Return type iterator

7.2 Session Module

The session module handles tasks that are associated with opening a session to an APIC or Fabric Node.

The session module contains two classes to open sessions with the APIC or Fabric Nodes:

1. LoginSession - uses a username and password to login
2. CertSession - uses a private key to generate signatures for every transaction, the user needs to have a X.509 certificate associated with their local user.

The LoginSession is the most robust method allowing access to both the APIC's and the Fabric Nodes (switches) and can support all methods of RBAC. The CertSession method of generating signatures is limited to only communicating with the APIC and can not support any form of RBAC. One other limitation of CertSession type of sesions is there is no support for eventchannel notifications.

To make changes to the APIC configuration using the Python API, you must use a user with write privileges. When using a LoginSession, once a user is authenticated, the API returns a data structure that includes a session timeout period in seconds and a token that represents the session. The token is also returned as a cookie in the HTTP response header. To maintain your session, you must send login refresh messages to the API within the session timeout period. The token changes each time that the session is refreshed.

The following sections describe the classes in the session module.

7.2.1 AbstractSession

Class that abstracts sessions. This is used by LoginSession and CertSession and should not be instantiated directly. Instead use one of the other session classes.

class cobra.mit.session.**AbstractSession**(*controllerUrl*, *secure*, *timeout*, *requestFormat*)

__init__ (*controllerUrl*, *secure*, *timeout*, *requestFormat*)
Initialize self. See help(type(self)) for accurate signature.

secure
verifies server authenticity

timeout
communication timeout for the connection

7.2.2 LoginSession

Class that creates a login session with a username and password.

Example of using a LoginSession:

```
from cobra.mit.access import MoDirectory
from cobra.mit.session import LoginSession

session = LoginSession('http://10.1.1.1', 'user', 'password', secure=False)
moDir = MoDirectory(session)
moDir.login()
allTenants = moDir.lookupByClass('fvTenant')
for tenant in allTenants:
    print(tenant.name)
```

class cobra.mit.session.**LoginSession**(*controllerUrl*, *user*, *password*, *secure=False*, *timeout=90*, *requestFormat='xml'*)
The LoginSession class creates a login session with a username and password

__init__(*controllerUrl*, *user*, *password*, *secure=False*, *timeout=90*, *requestFormat='xml'*)

Parameters

- **user** (*str*) – Username
- **password** (*str*) – Password

challenge
Authentication challenge for this session

cookie
Authentication cookie for this session

password
Returns the password.

refreshTime
Returns the relative login refresh time. The session must be refreshed by this time or it times out

refreshTimeoutSeconds
Returns the number of seconds for which this LoginSession is valid

user
Returns the username.

version
Returns APIC version received from aaaLogin

7.2.3 CertSession

Class that creates a unique token per URI path based on a signature created by a SSL. Locally this uses a private key to create that signature. On the APIC you have to already have provided a certificate with the users public key via the aaaUserCert class. This uses PyOpenSSL if it is available (install Cobra with the [ssl] option). If PyOpenSSL is not available this will try to fallback to openssl using subprocess and temporary files that should work for most platforms.

Steps to utilize CertSession

1. Create a local user on the APIC with a X.509 certificate in PEM format
2. Instantiate a CertSession class with the users certificate Dn and the private key
3. Make POST/GET requests using the Python SDK

Step 1: Create a local user with X.509 Certificate

The following is an example of how to use the Python SDK to configure a local user with a X.509 certificate. This is a required step and can be completed using the GUI, the REST API or the Python SDK. Once the local user exists and has a X.509 certificate attached to the local user, then the CertSession class can be used for that user.

```
# Generation of a certificate and private key using the subprocess module to
# make direct calls to openssl at the shell level.  This assumes that
# openssl is installed on the system.

from subprocess import Popen, CalledProcessError, PIPE
from cobra.mit.access import MoDirectory
from cobra.mit.session import LoginSession
from cobra.mit.request import ConfigRequest
from cobra.model.pol import Uni as PolUni
from cobra.model.aaa import UserEp as AaaUserEp
from cobra.model.aaa import User as AaaUser
from cobra.model.aaa import UserDomain as AaaUserDomain
from cobra.model.aaa import UserRole as AaaUserRole
from cobra.model.aaa import UserCert as AaaUserCert

certUser = 'myuser'
pKeyFile = 'myuser.key'
certFile = 'myuser.cert'

# Generate the certificate in the current directory
cmd = ["openssl", "req", "-new", "-newkey", "rsa:1024", "-days", "36500",
       "-nodes", "-x509", "-keyout", pKeyFile, "-out", certFile,
       "-subj", "/CN=Generic/O=Acme/C=US"]
proc = Popen(cmd, stdin=PIPE, stdout=PIPE, stderr=PIPE)
out, error = proc.communicate()
# If an error occurs, fail
if proc.returncode != 0:
    print("Output: {0}, Error {1}".format(out, error))
    raise CalledProcessError(proc.returncode, " ".join(cmd))

# At this point pKeyFile and certFile exist as files in the local directory.
# pKeyFile will be used when we want to generate signatures.  certFile is
# contains the X.509 certificate (with public key) that needs to be pushed
# to the APIC for a local user.
```

(continues on next page)

(continued from previous page)

```
with open(certFile, "r") as file:
    PEMdata = file.read()

# Generate a local user to commit to the APIC
polUni = PolUni('')
aaaUserEp = AaaUserEp(polUni)
aaaUser = AaaUser(aaaUserEp, certUser)
aaaUserDomain = AaaUserDomain(aaaUser, name='all')
# Other aaaUserRoles maybe needed to give the user other privileges
aaaUserRole = AaaUserRole(aaaUserDomain, name='read-all',
                          privType='readPriv')
# Attach the certificate to that user.
aaaUserCert = AaaUserCert(aaaUser, certUser + '-cert')
# Using the data read in from the certificate file.
aaaUserCert.data = PEMdata

# Push the new local user to the APIC
session = LoginSession('https://10.1.1.1', 'admin', 'ins3965!', secure=False)
moDir = MoDirectory(session)
moDir.login()
cr = ConfigRequest()
cr.addMo(aaaUser)
moDir.commit(cr)
```

Steps 2 and 3: Instantiate and use a CertSession class

This step requires you know two pieces of information:

1. The users certificate distinguished name (Dn)
2. The private key that was created at the time of the certificate

The private key should be kept secret to ensure the highest levels of security for this type of session.

The certificate Dn will be in the form of:

uni/userext/user-<userid>/usercert-<certName>

You can also use a aaaUserCert managed object to get this Dn - as in the example below. The following example shows how to query the APIC for all tentants using a CertSession:

```
from cobra.mit.access import MoDirectory
from cobra.mit.session import CertSession
from cobra.model.pol import Uni as PolUni
from cobra.model.aaa import UserEp as AaaUserEp
from cobra.model.aaa import User as AaaUser
from cobra.model.aaa import UserCert as AaaUserCert

certUser = 'myuser'
pKeyFile = 'myuser.key'

# Generate a local user object that matches the one on the APIC
# This is only being used to get the Dn of the user's certificate
polUni = PolUni('')
aaaUserEp = AaaUserEp(polUni)
aaaUser = AaaUser(aaaUserEp, certUser)
```

(continues on next page)

(continued from previous page)

```
# Attach the certificate to that user.
aaaUserCert = AaaUserCert(aaaUser, certUser + '-cert')

# Read in the private key data from a file in the local directory
with open(pKeyFile, "r") as file:
    pKey = file.read()

# Instantiate a CertSession using the dn and private key
session = CertSession('https://10.1.1.1', aaaUserCert.dn, pKey, secure=False)
moDir = MoDirectory(session)

# No login is required for certificate based sessions
allTenants = moDir.lookupByClass('fvTenant')
print(allTenants)
```

class cobra.mit.session.**CertSession**(*controllerUrl*, *certificateDn*, *privateKey*, *secure=False*, *timeout=90*, *requestFormat='xml'*)

The CertSession class creates a login session using a certificate dn and private key

__init__(*controllerUrl*, *certificateDn*, *privateKey*, *secure=False*, *timeout=90*, *requestFormat='xml'*)

Parameters **cert** (*str*) – Certificate String

certificateDn

Returns the certificate dn.

privateKey

Returns the private key.

7.3 Request Module

The request module handles configuration and queries to the APIC.

You can use the request module to:

- Create or update a managed object (MO)
- Call a method within an MO
- Delete an MO
- Run a query to read the properties and status of an MO or discover objects

7.3.1 Using Queries

Queries return information about an MO or MO properties within the APIC management information tree (MIT). You can apply queries that are based on a distinguished name (DN) and MO class.

7.3.2 Specifying a Query Scope

You can limit the scope of the response to an API query by applying scoping filters. You can limit the scope to the first level of an object or to one or more of its subtrees or children based on class, properties, categories, or qualification by a logical filter expression. This list describes the available scopes:

- self-(Default) Considers only the MO itself, not children or subtrees.

- children-Considers only the children of the MO, not the MO itself.
- subtree-Considers only the subtrees of the MO, not the MO itself.

7.3.3 Applying Query Filters

You can query on a variety of query filters, including:

- MO class
- Property
- Subtree
- Subtree and class

You can also include optional subtree values, including:

- audit-logs
- event-logs
- faults
- fault-records
- health
- health-records
- relations
- stats
- tasks
- count
- no-scoped
- required

7.3.4 Applying Configuration Requests

The request module handles configuration requests that are issued by the access module. The ConfigRequest class enables you to:

- Add an MO
- Remove an MO
- Verify if an MO is present in an uncommitted configuration
- Return the root MO for a given object

7.3.5 AbstractRequest

Class that represents an abstract request. AbstractQuery and ConfigRequest derive from this class.

class cobra.mit.request.**AbstractRequest**

AbstractRequest is the base class for all other request types, including AbstractQuery, ConfigRequest, UploadPackage, LoginRequest and RefreshRequest

__init__()
Initialize self. See help(type(self)) for accurate signature.

getUrl(*session*)
Returns the dn query URL containing all the query options defined on this object

id
Returns the id of this query if set, else None

classmethod makeOptions(*options*)
Returns a string containing the concatenated values of all key/value pairs for the options defined in dict options

options
Return the HTTP request query string string for this object

7.3.6 AbstractQuery

Class that represents an abstract query. ClassQuery and DnQuery derive from this class.

class cobra.mit.request.**AbstractQuery**
Class representing an abstract query. The class is used by classQuery and Dnquery.

__init__()
Initialize self. See help(type(self)) for accurate signature.

classFilter
Returns the current class filter type.

options
Returns the concatenation of the class and base class options for HTTP request query string

orderBy
Get the orderBy sort specifiers string.

Returns The order-by string of sort specifiers.

Return type str

page
Get the page value.

Returns The number of the page returned in the query.

Return type int

pageSize
Get the pageSize value.

Returns The number of results to be returned by a query.

Return type int

propFilter
Returns the current property filter type.

propInclude
Returns the current response property include filter

queryTarget
Returns the query type.

replica
Returns the current value for the replica option.

subtree
Returns the current type of subtree filter.

subtreeClassFilter
Returns the current subtree class filter.

subtreeInclude
Returns the current subtree query values.

subtreePropFilter
Returns the subtree prop filter.

7.3.7 DnQuery

Class that creates a query object based on distinguished name (DN).

class cobra.mit.request.**DnQuery**(*dn*)
Class to create a query based on distinguished name (Dn).

__eq__(*other*)
Implement ==.

__ge__(*other*)
Implement >=.

__gt__(*other*)
Implement >.

__hash__()
Return hash(self).

__init__(*dn*)

Parameters **dnStr** (*str*) – DN to query

__le__(*other*)
Implement <=.

__lt__(*other*)
Implement <.

__ne__(*other*)
Implement !=.

dnStr
Returns the base dnString for this DnQuery

options
Returns the concatenation of the class and base class options for HTTP request query string

7.3.8 ClassQuery

Class that creates a query object based on object class.

class cobra.mit.request.**ClassQuery**(*className*)
Class to create a query based on object class.

__eq__(*other*)
Implement ==.

__ge__(*other*)
Implement >=.

__gt__(*other*)
Implement >.

__hash__()
Return hash(self).

__init__(*className*)
Initialize self. See help(type(self)) for accurate signature.

__le__(*other*)
Implement <=.

__lt__(*other*)
Implement <.

__ne__(*other*)
Implement !=.

className
Returns the className targeted by this ClassQuery

options
Returns the concatenation of the class and base class options for HTTP request query string

7.3.9 ConfigRequest

Class that handles configuration requests. The *cobra.mit.access.MoDirectory.commit()* function uses this class.:

```
# Import the config request
from cobra.mit.request import ConfigRequest
configReq = ConfigRequest()
```

class cobra.mit.request.**ConfigRequest**
Class to handle configuration requests. The commit function uses this class.

__init__()
Initialize self. See help(type(self)) for accurate signature.

addMo(*mo*)
Adds a managed object (MO) to the configuration.

hasMo(*dn*)
Verifies whether managed object (MO) is present in an uncommitted configuration.

options
Returns the concatenation of the class and base class options for HTTP request query string

removeMo(*mo*)
Removes a managed object (MO) from the configuration.

subtree
Returns the current type of subtree filter.

7.3.10 Tag Request

Tags can be added to select MOs and become objects of type TagInst contained by that MO. Rather than having to instantiate an object of type tagInst and query for the containing MO, instantiate a tagInst object and add it to the containing MO then commit the whole thing, the REST API offers the ability to add one or more tags to a specific Dn using a specific API call. Cobra utilizes this API call in the TagsRequest class.

Tags can then be used to group or label objects and do quick and easy searches for objects with a specific tag using a normal ClassQuery with a property filter.

Tag queries allow you to provide a Dn and either a list of tags or a string (which should be comma separated in the form: tag1,tag2,tag3) for the add or remove properties. The class then builds the proper REST API queries as needed to add the tag(s) to the MO.

The class can also be used to do tag queries (HTTP GETs) against specific Dn's using the cobra.mit.access.MoDirectory.query() method with the cobra.mit.request.TagRequest instance provided as the query object.

Example Usage:

```
>>> from cobra.mit.session import LoginSession
>>> from cobra.mit.access import MoDirectory
>>> from cobra.mit.request import TagsRequest
>>> session = LoginSession('https://192.168.10.10', 'george', 'pa$sW0rd!',␣
↪secure=False)
>>> modir = MoDirectory(session)
>>> modir.login()
>>> tags = TagsRequest('uni/tn-common/ap-default')
>>> q = modir.query(tags)
>>> print q[0].name
pregnantSnake
>>> tags.remove = "pregnantSnake"
>>> modir.commit(tags)
<Response [200]>
>>> tags.add = ['That','is','1','dead','bird']
>>> modir.commit(tags)
<Response [200]>
>>> tags.add = "" ; tags.remove = []
>>> q = modir.query(tags)
>>> tags.remove = ','.join([rem.name for rem in q])
>>> print tags.remove
u'is,That,dead,bird,1'
>>> print tags.getUrl(session)
https://192.168.10.10/api/tag/mo/uni/tn-common/ap-default.json?remove=bird,1,is,That,
↪dead
>>> modir.commit(tags)
<Response [200]>
>>> modir.query(tags)
[]
>>>
```

class cobra.mit.request.**TagsRequest**(*dn*, *add=None*, *remove=None*)

Tags request to add or remove tags for a Dn.

__init__(*dn*, *add=None*, *remove=None*)

Parameters

- **dn** (*cobra.mit.naming.Dn or str*) – The Dn to do the Tags request against
- **add** (*str or list*) – The comma separated string or list of tags to add

- **remove** (*str or list*) – The comma separated string or list of tags to remove

add
Tags that will be added for this TagsRequest

Returns String form of the tags, comma separated

Return type str

dnStr
The Dn string for this request :rtype: str

Type returns

options
The url options string with & prepended :rtype: str

Type returns

remove
Tags that will be removed for this TagsRequest

Returns String form of the tags, comma separated

Return type str

7.3.11 TraceQuery

A class that creates a trace query

class cobra.mit.request.**TraceQuery**(*dn*, *targetClass*)
Class to create a trace query using base Dn and targetClass

__init__(*dn*, *targetClass*)
Initialize self. See help(type(self)) for accurate signature.

dnStr
Returns the base dnString for this DnQuery

options
Returns the concatenation of the class and base class options for HTTP request query string

targetClass
Returns the target class

7.4 Services Module

This module provides an interface to uploading L4-7 device packages to the controller. Refer to the **Developing L4-L7 Device Packages** document for more information on creating device packages.

Example:

```
session = cobra.mit.session.LoginSession('https://apic', 'admin',
                                         'password', secure=False)
moDir = cobra.mit.access.MoDirectory(session)
moDir.login()

packageUpload = cobra.services.UploadPackage('asa-device-pkg.zip')
response = moDir.commit(packageUpload)
```

The following sections describe the classes in the services module.

7.4.1 UploadPackage

Class for uploading L4-L7 device packages to APIC

class cobra.services.**UploadPackage**(*devicePackagePath*, *validate=False*)
Class for uploading L4-L7 device packages to APIC

__init__(*devicePackagePath*, *validate=False*)
Create an UploadPackage object that can be passed to MoDirectory.commit

Parameters

- **devicePackagePath** (*str*) – Path to the device package on the local file system
- **validate** (*bool*) – If true, the device package will be validated locally before attempting to upload

data
Returns the data this request should post

Returns string containing contents of device package

Return type str

devicePackagePath
Returns the currently configured path to the device package

Returns Path to the device package on the local file system

Return type str

getUrl(*session*)
Returns the URI this request will access

Parameters **session** (cobra.mit.session.AbstractSession) – session object for which the request will be sent

requestargs(*session*)
Returns the POST arguments for this request

Parameters **session** (cobra.mit.session.AbstractSession) – session object for which the request will be sent

Returns requests style kwargs that can be passed to request.post()

Return type dict

7.5 Access Module

The access module enables you to maintain network endpoints and manage APIC connections.

The following sections describe the classes in the access module.

7.5.1 MoDirectory

Class that creates a connection to the APIC and manage the MIT configuration. MoDirectory enables you to create queries based on the object class, distinguished name, or other properties, and to commit a new configuration. MoDirectory requires an existing session and endpoint.

class cobra.mit.access.**MoDirectory**(*session*)

The MoDirectory class creates a connection to the APIC and the MIT. MoDirectory requires an existing session and endpoint.

__init__(*session*)

Parameters **session** – Specifies a session

commit(*configObject*, *sync_wait_timeout=None*)

Short-form commit operation for a configRequest

exists(*dnStrOrDn*)

Checks if managed object (MO) with given distinguished name (dn) is present or not

Parameters **dnStrOrDn** (*str or cobra.mit.naming.Dn*) – A distinguished name as a *cobra.mit.naming.Dn* or string

Returns True, if MO is present, else False.

Return type bool

login()

Creates a session to an APIC.

logout()

Ends a session to an APIC.

lookupByClass(*classNames*, *parentDn=None*, ***queryParams*)

A short-form managed object (MO) query by class.

Parameters

- **classNames** – Name of the class to lookup
- **parentDn** – dn of the root object were to start search from (optional)
- **queryParams** – a dictionary including the properties to the added to the query.

lookupByDn(*dnStrOrDn*, ***queryParams*)

A short-form managed object (MO) query using the distinguished name(Dn) of the MO.

Parameters

- **dnStrOrDn** – dn of the object to lookup
- **queryParams** – a dictionary including the properties to the added to the query.

query(*queryObject*)

Queries the MIT for a specified object. The queryObject provides a variety of search options.

reauth()

Re-authenticate this session with the current authentication cookie. This method can be used to extend the validity of a successful login credentials. This method may fail if the current session expired on the server side. If this method fails, the user must login again to authenticate and effectively create a new session.

7.6 Managed Object (MO) Module

A Managed Object (MO) is an abstract representation of a physical or logical entity that contain a set of configurations and properties, such as a server, processor, or resource pool. The MO module represents MOs.

The APIC system configuration and state are modeled as a collection of managed objects (MOs). For example, servers, chassis, I/O cards, and processors are physical entities represented as MOs; resource pools, user roles, service profiles, and policies are logical entities represented as MOs.

7.6.1 Accessing Properties

When you create a managed object (MO), you can access properties as follows:

```
userMo = User('uni/userext', 'george')
userMo.firstName = 'George'
userMo.lastName = 'Washington'
```

7.6.2 Managing Properties

You can use the following methods to manage property changes on a managed object (MO):

- dirtyProps-Returns modified properties that have not been committed.
- isPropDirty-Indicates if there are unsaved changes to the MO properties.
- resetProps-Resets MO properties, discarding uncommitted changes.

7.6.3 Accessing Related Objects

The managed object (MO) object properties enable you to access related objects in the MIT using the following functions:

- parentDn-Returns the distinguished name (DN) of the parent managed object (MO).
- parent-Returns the parent MO.
- children-Returns the names of child MOs.
- numChildren-Returns the number of child MOs.

7.6.4 Verifying Object Status

You can use the status property to access the status of the Mo.

class cobra.mit.mo.**Mo**(*parentMoOrDn*, *markDirty*, **namingVals*, ***creationProps*)

A class to create managed objects (MOs), which represent a physical or logical entity with a set of configurations and properties.

__getattr__(*propName*)

Returns a managed object (MO) attribute.

__hash__()

Return hash(self).

__init__(*parentMoOrDn*, *markDirty*, **namingVals*, ***creationProps*)

Initialize self. See help(type(self)) for accurate signature.

__setattr__(*propName*, *propValue*)

Sets a managed object (MO) attribute.

children

Returns the child managed objects (MOs).

delete()

Marks the mo as deleted. If this mo is committed, the corresponding mo in the backend will be deleted.

dirtyProps

Returns modified properties that have not been committed.

dn
Returns the distinguished name (Dn) of the managed object (MO).

isPropDirty(*propName*)
Returns a value indicating whether a given property has a new value that has not been committed.

numChildren
Returns the number of child managed objects (MOs).

parent
Returns the parent managed object (MO).

parentDn
Returns the distinguished name (Dn) of the parent managed object (MO).

resetProps()
Resets managed object (MO) properties, discarding uncommitted changes.

rn
Returns the relative name (Rn) of the managed object (MO).

status
Returns the managed object (MO) status.

7.7 Meta Module

The following sections describe the classes in the meta module.

7.7.1 Category

Class that represents an object category.

class cobra.mit.meta.**Category**(*name*, *categoryId*)

__eq__(*other*)
Implement ==.

__ge__(*other*)
Implement >=.

__gt__(*other*)
Implement >.

__hash__()
Return hash(self).

__init__(*name*, *categoryId*)
Initialize self. See help(type(self)) for accurate signature.

__le__(*other*)
Implement <=.

__lt__(*other*)
Implement <.

__ne__(*other*)
Implement !=.

__str__()
Return str(self).

7.7.2 ClassLoader

Class that loads a specified class.

class cobra.mit.meta.**ClassLoader**

7.7.3 ClassMeta

Class that provides information about an object class.

class cobra.mit.meta.**ClassMeta**(*className*)

__init__(*className*)
Initialize self. See help(type(self)) for accurate signature.

7.7.4 Constant

class cobra.mit.meta.**Constant**(*const*, *label*, *value*)

__eq__(*other*)
Implement ==.

__ge__(*other*)
Implement >=.

__gt__(*other*)
Implement >.

__init__(*const*, *label*, *value*)
Initialize self. See help(type(self)) for accurate signature.

__le__(*other*)
Implement <=.

__lt__(*other*)
Implement <.

__ne__(*other*)
Implement !=.

__str__()
Return str(self).

7.7.5 NamedSourceRelationMeta

class cobra.mit.meta.**NamedSourceRelationMeta**(*className*, *targetClassName*)

__init__(*className*, *targetClassName*)
Initialize self. See help(type(self)) for accurate signature.

7.7.6 PropMeta

class cobra.mit.meta.**PropMeta**(*typeClassName*, *name*, *moPropName*, *propId*, *category*)

__eq__(*other*)
Implement ==.

__ge__(*other*)
Implement >=.

__gt__(*other*)
Implement >.

__hash__()
Return hash(self).

__init__(*typeClassName*, *name*, *moPropName*, *propId*, *category*)
Initialize self. See help(type(self)) for accurate signature.

__le__(*other*)
Implement <=.

__lt__(*other*)
Implement <.

__ne__(*other*)
Implement !=.

__str__()
Return str(self).

7.7.7 SourceRelationMeta

class cobra.mit.meta.**SourceRelationMeta**(*className*, *targetClassName*)

__init__(*className*, *targetClassName*)
Initialize self. See help(type(self)) for accurate signature.

7.7.8 TargetRelationMeta

class cobra.mit.meta.**TargetRelationMeta**(*className*, *sourceClassName*)

__init__(*className*, *sourceClassName*)
Initialize self. See help(type(self)) for accurate signature.

CHAPTER 8

Examples

8.1 Before You Begin

Before applying these examples, refer to the APIC documentation to understand the Cisco Application Centric Infrastructure (ACI) and the APIC. The APIC documentation contains explanations and examples of these and other tasks using the APIC GUI, CLI, and REST API. See the *Cisco APIC Getting Started Guide* for detailed examples.

8.2 Initial Statements for All Examples

The following setup statements or their equivalents are assumed to be present in any APIC Python API program using these code examples.

```
from cobra.mit.access import MoDirectory
from cobra.mit.session import LoginSession
session = LoginSession('https://sample-host.coolapi.com', 'admin',
                       'xxx?xxx?xxx')
moDir = MoDirectory(session)
moDir.login()
```

The above code snippet creates an **MoDirectory**, connects it to the endpoint and then performs authentication. The **moDir** can be used to query, create/delete Mos from the end point.

8.3 Creating a Tenant

The tenant (fv:Tenant object) is a container for policies that enable an administrator to exercise domain based access control so that qualified users can access privileges such as tenant administration and networking administration. According to the *Cisco APIC Management Information Model Reference*, an object of the fv:Tenant class is a child of the policy resolution universe ('uni') class. This example creates a tenant named 'ExampleCorp' under the 'uni' object.

```
# Import the config request
from cobra.mit.request import ConfigRequest
configReq = ConfigRequest()

# Import the tenant class from the model
from cobra.model.fv import Tenant

# Get the top level policy universe directory
uniMo = moDir.lookupByDn('uni')

# create the tenant object
fvTenantMo = Tenant(uniMo, 'ExampleCorp')
```

The command creates an object of the fv.Tenant class and returns a reference to the object. A tenant contains primary elements such as filters, contracts, bridge domains and application network profiles that we will create in later examples.

8.4 Application Profiles

An application profile (fv.Ap object) is a tenant policy that defines the policies, services, and relationships between endpoint groups (EPGs) within the tenant. The application profile contains EPGs that are logically related to one another. This example defines a web application profile under the tenant.

```
# Import the Ap class from the model
from cobra.model.fv import Ap

fvApMo = Ap(fvTenantMo, 'WebApp')
```

8.5 Endpoint Groups

An endpoint group is a collection of network-connected devices, such as clients or servers, that have common policy requirements. This example creates a web application endpoint group named 'WebEPG' that is contained in an application profile under the tenant.

```
# Import the AEPg class from the model
from cobra.model.fv import AEPg

fvAEPgMoWeb = AEPg(fvApMo, 'WebEPG')
```

8.6 Physical Domains

This example associates the web application endpoint group with a bridge domain.

```
# Import the related classes from the model
from cobra.model.fv import RsBd, Ctx, BD, RsCtx

# create a private network
fvCtxMo = Ctx(fvTenantMo, 'private-net1')
```

(continues on next page)

(continued from previous page)

```
# create a bridge domain
fvBDMo = BD(fvTenantMo, 'bridge-domain1')

# create an association of the bridge domain to the private network
fvRsCtx = RsCtx(fvBDMo, tnFvCtxName=fvCtxMo.name)

# create a physical domain associated with the endpoint group
fvRsBd1 = RsBd(fvAEPgMoWeb, fvBDMo.name)
```

8.7 Contracts and Filters

A contract defines the protocols and ports on which a provider endpoint group and a consumer endpoint group are allowed to communicate. You can use the **directory.create** function to define a contract, add a subject, and associate the subject and a filter.

This example creates a Web filter for HTTP (TCP port 80) traffic.

```
# Import the Filter and related classes from model
from cobra.model.vz import Filter, Entry, BrCP, Subj, RsSubjFiltAtt

# create a filter container (vz.Filter object) within the tenant
filterMo = Filter(fvTenantMo, 'WebFilter')

# create a filter entry (vz.Entry object) that specifies bidirectional
# HTTP (TCP/80) traffic
entryMo = Entry(filterMo, 'HttpPort')
entryMo.dFromPort = 80    # HTTP port
entryMo.dToPort = 80
entryMo.prot = 6          # TCP protocol number
entryMo.etherT = "ip"     # EtherType

# create a binary contract (vz.BrCP object) container within the
# tenant
vzBrCPMoHTTP = BrCP(fvTenantMo, 'WebContract')

# create a subject container for associating the filter with the
# contract
vzSubjMo = Subj(vzBrCPMoHTTP, 'WebSubject')
RsSubjFiltAtt(vzSubjMo, tnVzFilterName=filterMo.name)
```

8.8 Namespaces

A namespace identifies a range of traffic encapsulation identifiers for a VMM domain or a VM controller. A namespace is a shared resource and can be consumed by multiple domains such as VMM and L4-L7 services. This example creates and assigns properties to a VLAN namespace.

```
# Import the namespaces related classes from model
from cobra.model.fvns import VlanInstP, EncapBlk

fvnsVlanInstP = VlanInstP('uni/infra', 'namespace1', 'dynamic')
fvnsEncapBlk = EncapBlk(fvnsVlanInstP, 'vlan-5', 'vlan-20',
```

(continues on next page)

(continued from previous page)

```
                              name='encap')
nsCfg = ConfigRequest()
nsCfg.addMo(fvnsVlanInstP)
moDir.commit(nsCfg)
```

8.9 VM Networking

This example creates a virtual machine manager (VMM) and configuration.

```
# Import the namespaces related classes from model
from cobra.model.vmm import ProvP, DomP, UsrAccP, CtrlrP, RsAcc
from cobra.model.infra import RsVlanNs

vmmProvP = ProvP('uni', 'VMWare')
vmmDomP = DomP(vmmProvP, 'Datacenter')
vmmUsrAccP = UsrAccP(vmmDomP, 'default', pwd='password', usr='administrator')
vmmRsVlanNs = RsVlanNs(vmmDomP, fvnsVlanInstP.dn)
vmmCtrlrP = CtrlrP(vmmDomP, 'vserver-01', hostOrIp='192.168.64.9')
vmmRsAcc = RsAcc(vmmCtrlrP, tDn=vmmUsrAccp.dn)

# Add the tenant object to the config request and commit
confgReq.addMo(fvTenantMo)
moDir.commit(configReq)
```

8.10 Creating a Complete Tenant Configuration

This example creates a tenant named 'ExampleCorp' and deploys a three-tier application including Web, app, and database servers. See the similar three-tier application example in the *Cisco APIC Getting Started Guide* for additional description of the components being configured.

```
1  from __future__ import print_function
2  # Copyright 2015 Cisco Systems, Inc.
3  #
4  # Licensed under the Apache License, Version 2.0 (the "License");
5  # you may not use this file except in compliance with the License.
6  # You may obtain a copy of the License at
7  #
8  #      http://www.apache.org/licenses/LICENSE-2.0
9  #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15
16 #!/usr/bin/env python
17
18
19 # Import access classes
20 from cobra.mit.access import MoDirectory
21 from cobra.mit.session import LoginSession
```

(continues on next page)

(continued from previous page)

```
from cobra.mit.request import ConfigRequest

# Import model classes
from cobra.model.fvns import VlanInstP, EncapBlk
from cobra.model.infra import RsVlanNs
from cobra.model.fv import Tenant, Ctx, BD, RsCtx, Ap, AEPg, RsBd, RsDomAtt
from cobra.model.vmm import DomP, UsrAccP, CtrlrP, RsAcc


# Policy information
VMM_DOMAIN_INFO = {'name': "mininet",
                   'ctrlrs': [{'name': 'vcenter1', 'ip': '192.0.20.3',
                               'scope': 'vm'}],
                   'usrs': [{'name': 'admin', 'usr': 'administrator',
                             'pwd': 'pa$$word1'}],
                   'namespace': {'name': 'VlanRange', 'from': 'vlan-100',
                                 'to': 'vlan-200'}
                  }

TENANT_INFO = [{'name': 'ExampleCorp',
                'pvn': 'pvn1',
                'bd': 'bd1',
                'ap': [{'name': 'OnlineStore',
                        'epgs': [{'name': 'app'},
                                 {'name': 'web'},
                                 {'name': 'db'},
                                ]
                       },
                      ]
                }
               ]

def main(host, port, user, password):

    # CONNECT TO APIC
    print('Initializing connection to APIC...')
    apicUrl = 'http://%s:%d' % (host, port)
    moDir = MoDirectory(LoginSession(apicUrl, user, password))
    moDir.login()

    # Get the top level Policy Universe Directory
    uniMo = moDir.lookupByDn('uni')
    uniInfraMo = moDir.lookupByDn('uni/infra')

    # Create Vlan Namespace
    nsInfo = VMM_DOMAIN_INFO['namespace']
    print("Creating namespace %s.." % (nsInfo['name']))
    fvnsVlanInstPMo = VlanInstP(uniInfraMo, nsInfo['name'], 'dynamic')
    #fvnsArgs = {'from': nsInfo['from'], 'to': nsInfo['to']}
    EncapBlk(fvnsVlanInstPMo, nsInfo['from'], nsInfo['to'], name=nsInfo['name'])

    nsCfg = ConfigRequest()
    nsCfg.addMo(fvnsVlanInstPMo)
    moDir.commit(nsCfg)

    # Create VMM Domain
    print('Creating VMM domain...')
```

(continues on next page)

(continued from previous page)

```
79
80      vmmpVMwareProvPMo = moDir.lookupByDn('uni/vmmp-VMware')
81      vmmDomPMo = DomP(vmmpVMwareProvPMo, VMM_DOMAIN_INFO['name'])
82
83      vmmUsrMo = []
84      for usrp in VMM_DOMAIN_INFO['usrs']:
85          usrMo = UsrAccP(vmmDomPMo, usrp['name'], usr=usrp['usr'],
86                          pwd=usrp['pwd'])
87          vmmUsrMo.append(usrMo)
88
89      # Create Controllers under domain
90      for ctrlr in VMM_DOMAIN_INFO['ctrlrs']:
91          vmmCtrlrMo = CtrlrP(vmmDomPMo, ctrlr['name'], scope=ctrlr['scope'],
92                              hostOrIp=ctrlr['ip'])
93          # Associate Ctrlr to UserP
94          RsAcc(vmmCtrlrMo, tDn=vmmUsrMo[0].dn)
95
96      # Associate Domain to Namespace
97      RsVlanNs(vmmDomPMo, tDn=fvnsVlanInstPMo.dn)
98
99      vmmCfg = ConfigRequest()
100     vmmCfg.addMo(vmmDomPMo)
101     moDir.commit(vmmCfg)
102     print("VMM Domain Creation Completed.")
103
104     print("Starting Tenant Creation..")
105     for tenant in TENANT_INFO:
106         print("Creating tenant %s.." % (tenant['name']))
107         fvTenantMo = Tenant(uniMo, tenant['name'])
108
109         # Create Private Network
110         Ctx(fvTenantMo, tenant['pvn'])
111
112         # Create Bridge Domain
113         fvBDMo = BD(fvTenantMo, name=tenant['bd'])
114
115         # Create association to private network
116         RsCtx(fvBDMo, tnFvCtxName=tenant['pvn'])
117
118         # Create Application Profile
119         for app in tenant['ap']:
120             print('Creating Application Profile: %s' % app['name'])
121             fvApMo = Ap(fvTenantMo, app['name'])
122
123             # Create EPGs
124             for epg in app['epgs']:
125
126                 print("Creating EPG: %s..." % (epg['name']))
127                 fvAEPgMo = AEPg(fvApMo, epg['name'])
128
129                 # Associate EPG to Bridge Domain
130                 RsBd(fvAEPgMo, tnFvBDName=tenant['bd'])
131                 # Associate EPG to VMM Domain
132                 RsDomAtt(fvAEPgMo, vmmDomPMo.dn)
133
134         # Commit each tenant seperately
135         tenantCfg = ConfigRequest()
```

(continues on next page)

(continued from previous page)

```
        tenantCfg.addMo(fvTenantMo)
        moDir.commit(tenantCfg)
    print('All done!')

if __name__ == '__main__':
    from argparse import ArgumentParser
    parser = ArgumentParser("Tenant creation script")
    parser.add_argument('-d', '--host', help='APIC host name or IP',
                        required=True)
    parser.add_argument('-e', '--port', help='server port', type=int,
                        default=80)
    parser.add_argument('-p', '--password', help='user password',
                        required=True)
    parser.add_argument('-u', '--user', help='user name', required=True)
    args = parser.parse_args()

    main(args.host, args.port, args.user, args.password)

```

8.11 Creating a Query Filter

This example creates a query filter property to match fabricPathEpCont objects whose nodeId property is 101.

```
# Import the related classes from model
from cobra.model.fabric import PathEpCont

nodeId = 101
myClassQuery.propFilter = 'eq(fabricPathEpCont.nodeId, "{0}")'.format(nodeId)
```

The basic filter syntax is 'condition(item1, "value")'. To filter on the property of a class, the first item of the filter is of the form pkgClass.property. The second item of the filter is the property value to match. The quotes are necessary.

8.12 Accessing a Child MO

This example shows how to access a child MO, such as a bridge-domain, which is a child object of a tenant MO.

```
dnQuery = DnQuery('uni/tn-coke')
dnQuery.subtree = 'children'
tenantMo = moDir.query(dnQuery)
defaultBDMo = tenantMo.BD['default']
```

8.13 Iteration for a Child MO

This example shows how to user iteration for a child MO.

```
dnQuery = DnQuery('uni/tn-coke')
dnQuery.subtree = 'children'
tenantMo = moDir.query(dnQuery)
for bdMo in tenantMo.BD:
    print str(bdMo.dn)
```

CHAPTER 9

Tools for API Development

To create API commands and perform API functions, you must determine which MOs and properties are related to your task, and you must compose data structures that specify settings and actions on those MOs and properties. Several resources are available for that purpose.

9.1 APIC Management Information Model Reference

The *Cisco APIC Management Information Model Reference* is a Web-based tool that lists all object classes and their properties. The reference also provides the hierarchical structure, showing the ancestors and descendants of each object, and provides the form of the distinguished name (DN) for an MO of a class.

9.2 API Inspector

The API Inspector is a built-in tool of the APIC graphical user interface (GUI) that allows you to capture internal REST API messaging as you perform tasks in the APIC GUI. The captured messages show the MOs being accessed and the JSON data exchanges of the REST API calls. You can use this data when designing Python API calls to perform similar functions.

You can find instructions for using the API Inspector in the *Cisco APIC REST API User Guide*.

9.3 Browsing the Management Information Tree With the CLI

The APIC command-line interface (CLI) represents the management information tree (MIT) in a hierarchy of directories, with each directory representing a managed object (MO). You can browse the directory structure by doing the following:

1. Open an SSH session to the APIC to reach the CLI
2. Go to the directory /mit

For more information on the APIC CLI, see the *Cisco APIC Command Reference*.

9.4 Managed Object Browser (Visore)

The Managed Object Browser, or Visore, is a utility built into the APIC that provides a graphical view of the managed objects (MOs) using a browser. The Visore utility uses the APIC REST API query methods to browse MOs active in the Application Centric Infrastructure Fabric, allowing you to see the query that was used to obtain the information. The Visore utility cannot be used to perform configuration operations.

You can find instructions for using the Managed Object Browser in the *Cisco APIC REST API User Guide*.

9.5 APIC Getting Started Guide

The *Cisco APIC Getting Started Guide* contains many detailed examples of APIC configuration tasks using the APIC GUI, CLI, and REST API.

CHAPTER 10

Frequently Asked Questions

The following sections provide troubleshooting tips for common problems when using the APIC Python API.

10.1 Authentication Error

Ensure that you have the correct login credentials and that you have created a MoDirectory MO.

10.2 Inactive Configuration

If you have modified the APIC configuration and the new configuration is not active, ensure that you have committed the new configuration using the **MoDirectory.commit** function.

10.3 Keyword Error

To use a reserved keyword, from the API, include the _ suffix. In the following example, from is translated to from_:

```
def __init__(self, parentMoOrDn, from_, to, **creationProps):
    namingVals = [from_, to]
    Mo.__init__(self, parentMoOrDn, *namingVals, **creationProps)
```

10.4 Name Error

If you see a NameError for a module, such as cobra or access, ensure that you have included an import statement in your code such as:

```
import cobra
from cobra.mit import access
```

10.5 Python Path Errors

Ensure that your PYTHONPATH variable is set to the correct location. For more information, refer to http://www.python.org. You can use the **sys.path.append** python function or set PYTHONPATH environment variable to append a directory to your Python path.

10.6 Python Version Error

The APIC Python API is supported with versions 2.7 and 3.4 of Python.

10.7 WindowsError

If you see a **WindowsError: [Error 2] The system cannot find the file specified,** when trying to use the CertSession class, it generally means that you do not have openssl installed on Windows. Please see *Installing the Cisco APIC Python SDK*

10.8 ImportError for cobra.mit.meta.ClassMeta

If you see an **ImportError: No module named mit.meta** when trying to import something from the cobra.model namepsace, ensure that you have the acicobra package installed. Please see *Installing the Cisco APIC Python SDK*

10.9 ImportError for cobra.model.*

If you see an **ImportError: No module named model.** when importing anything from the cobra.model namespace, ensure that you have the acimodel package installed. Please see *Installing the Cisco APIC Python SDK*

CHAPTER 11

Download Cobra SDK

- ACI Cobra Runtime/SDK & Model

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

C

D

E

F

G

H

I

L

M

N