Cisco APIC Python API Documentation

Release 0.1

May 31, 2017

Contents

Contents:

CHAPTER 1

Understanding the Cisco Application Policy Infrastructure Controller

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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>'

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.

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

Installation Requirements:

The Cisco APIC Python SDK ("cobra") comes in two installable .egg 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 .egg files from a running instance of APIC at this URL:

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

The /cobra/_downloads directory contains the two .egg 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-1.1_1j-py2.7.egg
    acimodel-1.1_1j-py2.7.egg
```

In this example, each .egg filename references the APIC version 1.1(1j) from which it was created and the Python version py2.7 with which it is compatible.

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 - For more information, see https://www.python.org/.
- easy_install - For more information about easy_install, see https://pypi.python.org/pypi/setuptools.
- 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.

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 .egg files to your development system.

4. Install the egg file using the following command:

   From a local directory (relative or absolute):

   ```
   easy_install -Z *directory/path*/acicobra
   ```

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

   ```
   $ easy_install -Z ./cobra-eggs/acicobra-1.1_1j-py2.7.egg
   ```

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

   ```
   easy_install --user -Z *directory/path*/acicobra
   ```

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

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 egg123
```

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

8. Upgrade pip in the virtual environment.

```
c:\users\username\Envs\egg123
python -m pip install --upgrade pip
```

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

   From a local directory (relative or absolute):

```
easy_install -Z \*directory\path*\acicobra
```

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

```
> easy_install -Z cobra-eggs\acicobra-1.1_1j-py2.7.egg
```

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

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

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 easy_install:

```
easy_install -Z *directory/path*/acimodel-*version*-py2.7.egg
```

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

```
easy_install -Z ./cobra-eggs/acimodel-1.1_1j-py2.7.egg
```

Note: The .egg 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: 1.1_1j
Location: /local/lib/python2.7/site-packages/acimodel-1.1_1j-py2.7.egg
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.

Installing pyopenssl on Unix and Linux

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

1. Install the SDK .egg file using the following command:

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

   ```
   easy_install -Z --find-links *directory/path*/acicobra[ssl]
   ```

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

   ```
   $ easy_install -Z --find-links ./cobra-eggs/acicobra-1.1_1j-py2.7.egg[ssl]
   ```

Installing pyopenssl on Windows

In *Installing the SDK on Windows*, substitute the following procedure for the step where the SDK .egg file is installed. If you have created a virtual environment for the SDK, enter these commands 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 .egg file using the following command:

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

```
easy_install -Z --find-links *directory\path*\acicobra[ssl]
```

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

```
> easy_install -Z --find-links cobra-eggs\acicobra-1.1_1j-py2.7.egg[ssl]
```

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.

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
```

Path Settings

If you installed the cobra sdk egg 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')
```

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.

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.

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.

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)
```

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:

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.

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 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.

namingVals

tupleiterator – An interator for the naming values - readonly

meta

cobra.mit.meta.ClassMeta – The class meta for this Rn - readonly

moClass

cobra.mit.mo.Mo – The class of the Mo for this Rn - readonly

__eq__(*other*)

Implement ==.

__ge__(*other*)

Implement >=.

__gt__(*other*)

Implement >.

__init__(*classMeta*, **namingVals*)

Initalize a Rn object.

Parameters

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

__le__(*other*)

Implement <=.

__lt__(*other*)

Implement <.

__ne__(*other*)

Implement !=.

classmethod fromString(*classMeta*, *rnStr*)

Create a relative name instance from a string and classMeta.

Parameters

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

Raises ValueError – If the Rn prefix is not valid or the Rn does not follow the proper rnFormat

Returns The Rn object

Return type *cobra.mit.naming.Rn*

meta
Get the meta object for this Rn.

Returns The meta object for this Rn.

Return type *cobra.mit.meta.ClassMeta*

moClass
Get the Mo class from the meta for this Rn.

Returns The Mo class from the meta for this Rn.

Return type *cobra.mit.mo.Mo*

namingVals
Get the naming vals for this Rn as an iterator.

Returns The naming vals for this Rn.

Return type iterator

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*)
A Distinguised name class.

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"

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

meta
cobra.mit.meta.ClassMeta – class meta of the mo class for this Dn

moClass
cobra.mit.mo.Mo – Mo class for this Dn

contextRoot
cobra.mit.mo.Mo – The context root for this Dn

__eq__(*other*)
Implement ==.

__ge__(*other*)
Implement >=.

__gt__(*other*)
Implement >.

__init__(*rns=None*)
Initialize a Dn instance from list of Rn objects.

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

__le__(*other*)
Implement <=.

__lt__(*other*)
Implement <.

__ne__(*other*)
Implement !=.

appendRn(*rn*)
Append an Rn to this Dn.

Note: This changes the target MO

Parameters **rn** (cobra.mit.naming.Rn) – The Rn to append to this Dn

Raises ValueError – If the Dn can not contain the Rn

clone()
Get a new copy of this Dn.

Returns Copy of this Dn

Return type *cobra.mit.naming.Dn*

contextRoot
Get the Dn's context root.

Returns If the Dn has no context root. cobra.mit.meta.ClassMeta: The class meta for this Dn's Rn.

Return type None

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

Parameters **dns** (*list*) – The Dn objects to find the common parent of

Returns

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

Return type *cobra.mit.naming.Dn*

classmethod **fromString**(*dnStr*)
Create a distingushed name instance from a dn string.

Parses the dn string into its constituent Rn strings and creates the Rn objects.

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

Raises ValueError – If an Rn in the Dn is found to not be consistent with the ACI model

Returns (cobra.mit.naming.Dn): The Dn instance

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

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

Returns The Dn object of the ancestor as specified by the level argument

Return type *cobra.mit.naming.Dn*

getParent()
Get the parent Dn of the current Dn.

Same as:

self.getAncetor(1)

Returns Dn object of the immediate parent

Return type *cobra.mit.naming.Dn*

isAncestorOf(*descendantDn*)
Check if a Dn is an ancestor of this Dn.

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

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

Return type bool

isDescendantOf(*ancestorDn*)
Check if a Dn is a descendant of this Dn.

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

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

Return type boo

meta
Get the meta object for this Dn.

Returns The class meta for this Dn.

Return type *cobra.mit.meta.ClassMeta*

moClass
Get the Mo class for this Dn.

Returns The Mo class for this Dn.

Return type *cobra.mit.mo.Mo*

rn(*index=None*)
Get a Rn at a specified index.

If index is None, then the Rn of the target Mo is returned

Parameters **index** (*None or int*) – index of the Rn object, this must be between 0 and the length of the Dn (i.e. number of Rns) or None. The default is None

Returns (cobra.mit.naming.Rn): Rn object at the specified index

rns
Get the Rn's that make up this Dn as an iterator.

Returns An iterator object reprsenting the Rn's for this Dn.

Return type iterator

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.

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*)
Abstract session class.

Other sessions classes should derive from this class.

secure
bool – Only used for https. If True the remote server will be verified for authenticity. If False the remote server will not be verified for authenticity - readonly

timeout
int – Request timeout - readonly

url
str – The APIC or fabric node URL - readonly

formatType
str – The format type for the request - readonly

formatStr
str – The format string for the request, either xml or json - readonly

__init__(*controllerUrl*, *secure*, *timeout*, *requestFormat*)
Initialize an AbstractSession instance.

Parameters

- **controllerURL** (*str*) – The URL to reach the controller or fabric node
- **secure** (*bool*) – Only used for https. If True the remote server will be verified for authenticity. If False the remote server will not be verified for authenticity.
- **timeout** (*int*) – Request timeout
- **requestFormat** (*str*) – The format to send the request in. Valid values are xml or json.

Raises NotImplementedError – If the requestFormat is not valid

codec
Get the codec being used for this session.

Returns The codec being used for this session.

Return type cobra.mit.codec.AbstractCodec

formatStr
Get the format string for this session.

Returns

The formatType represented as a string. Currently this is either 'xml' or 'json'.

Return type str

formatType
Get the format type for this session.

Returns The format type represented as an integer

Return type int

get(*queryObject*)
Perform a query using the specified queryObject.

Parameters **queryObject** (cobra.mit.request.AbstractQuery) – The query object to use for the query.

Returns The query response parsed into a managed object

Return type *cobra.mit.mo.Mo*

login()
Login to the remote server.

A generic login method that should be overridden by classes that derive from this class

logout()
Logout from the remote server.

A generic logout method that should be overridden by classes that derive from this class

post(*requestObject*)
Perform a request using the specified requestObject.

Parameters **requestObject** (cobra.mit.request.AbstractRequest) – The request object to use for the request.

Returns The raw requests response.

Return type requests.response

refresh()
Refresh the session to the remote server.

A generic refresh method that should be overridden by classes that derive from this class

secure
Get the secure value.

Returns

True if the certificate for remote device should be verified, False otherwise.

Return type bool

timeout
Get the request timeout value.

Returns The time a request is allowed to take before an error is raised.

Return type int

url
Get the URL for the remote system.

Returns The URl for the remote system.

Return type str

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'*)
A login session with a username and password.

Note: The username and password are stored in memory.

user
str – The username to use for this session - readonly

password
str – The password to use for this session - readonly

cookie
str or None – The authentication cookie string for this session

challenge
str or None – The authentication challenge string for this session

version
str or None – The APIC software version returned once successfully logged in - readonly

refreshTime
str or None – The relative login refresh time. The session must be refreshed by this time or it times out - readonly

refreshTimeoutSeconds
str or None – The number of seconds for which this session is valid - readonly

domains
list – A list of possible login domains. The list is only populated once getLoginDomains() is called and this method can be called prior to logging in.

loginDomain
str – The login domain that should be used to login to the remote device. This is used to build a username that uses the loginDomain.

banner
str – The banner set on the APIC. This is set when the getLoginDomains() method is called.

secure
bool – Only used for https. If True the remote server will be verified for authenticity. If False the remote server will not be verified for authenticity - readonly

timeout
int – Request timeout - readonly

url
str – The APIC or fabric node URL - readonly

formattype
str – The format type for the request - readonly

formatStr
str – The format string for the request, either xml or json - readonly

__init__(*controllerUrl*, *user*, *password*, *secure=False*, *timeout=90*, *requestFormat='xml'*)
Initialize a LoginSession instance.

Parameters

- **controllerURL** (*str*) – The URL to reach the controller or fabric node
- **user** (*str*) – The username to use to authenticate
- **password** (*str*) – The password to use to authenticate
- **secure** (*bool*) – Only used for https. If True the remote server will be verified for authenticity. If False the remote server will not be verified for authenticity.
- **timeout** (*int*) – Request timeout

- **requestFormat** (*str*) – The format to send the request in. Valid values are xml or json.

banner
Get the banner.

Returns

The banner or an empty string if the getLoginDomains method has not been called.

Return type str

challenge
Get the challenge key value.

Returns The challeng key value.

Return type str

cookie
Get the session cookie value.

Returns The value of the session cookie.

Return type str

domains
Get the session login domains.

Returns The list of login domains.

Return type list

getHeaders(*uriPathAndOptions*, *data*)
Get the HTTP headers for a given URI path and options string.

Parameters

- **uriPathAndOptions** (*str*) – The full URI path including the options string
- **data** (*str*) – The payload

Returns The headers for this session class

Return type dict

getLoginDomains()
Get the possible login domains prior to login.

The domains are returned as a list.

login()
Login in to the remote server (APIC or Fabric Node).

Raises LoginError – If there was an error during login or the response could not be parsed.

loginDomain
Get the loginDomain.

Returns The loginDomain.

Return type str

logout()
Logout of the remote server (APIC or Fabric Node).

Currently this method does nothing

password
Get the password being used for this session.

Returns The session password.

Return type str

refresh()
Refresh a session with the remote server (APIC or Fabric Node).

Raises `LoginError` – If there was an error when refreshing the session or the response could not be parsed.

refreshTime
Get the refresh time.

Returns The refresh time returned by the login request.

Return type int

refreshTimeoutSeconds
Get the refresh timeout in seconds.

Returns The refresh timeout in seconds returned by the login request.

Return type int

user
Get the username being used for this session.

This can not be changed. If you need to change the session username, instantiate a new session object.

If the loginDomain is set, the username is set to:

apic:<loginDomain>\<user>

Returns The username for this session.

Return type str

version
Get the version.

Returns The version returned by the login request.

Return type str

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.

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)
# 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'*)

A session using a certificate dn and private key to generate signatures.

certificateDn
str – The distingushed name (Dn) for the users X.509 certificate - readonly

privateKey
str – The private key to use when calculating signatures. Must be paired with the private key in the X.509 certificate - readonly

cookie
str or None – The authentication cookie string for this session

challenge
str or None – The authentication challenge string for this session

version
str or None – The APIC software version returned once successfully logged in - readonly

refreshTime
str or None – The relative login refresh time. The session must be refreshed by this time or it times out - readonly

refreshTimeoutSeconds
str or None – The number of seconds for which this session is valid - readonly

secure
bool – Only used for https. If True the remote server will be verified for authenticity. If False the remote server will not be verified for authenticity - readonly

timeout
int – Request timeout - readonly

url
str – The APIC or fabric node URL - readonly

formattype
str – The format type for the request - readonly

formatStr
str – The format string for the request, either xml or json - readonly

__init__(*controllerUrl*, *certificateDn*, *privateKey*, *secure=False*, *timeout=90*, *requestFormat='xml'*)
Initialize a CertSession instance.

Parameters

- **controllerURL** (*str*) – The URL to reach the controller or fabric node
- **certificateDn** (*str*) – The distinguished name of the users certificate
- **privateKey** (*str*) – The private key to be used to calculate a signature
- **secure** (*bool*) – Only used for https. If True the remote server will be verified for authenticity. If False the remote server will not be verified for authenticity.
- **timeout** (*int*) – Request timeout
- **requestFormat** (*str*) – The format to send the request in. Valid values are xml or json.

certificateDn
Get the certificateDn for the user for this session.

Returns The certifcate Dn for this session.

Return type str

getHeaders(*uriPathAndOptions*, *data*)
Get the HTTP headers for a given URI path and options string.

Parameters

- **uriPathAndOptions** (*str*) – The full URI path including the options string
- **data** (*str*) – The payload

Returns The headers for this session class

Return type dict

getLoginDomains()
The getLoginDomains method.

Not (yet) relevant for CertSession but is included for consistency.

login()
login method.

Not relevant for CertSession but is included for consistency.

logout()
logout method.

Not relevant for CertSession but is included for consistency.

privateKey
Get the private key for this session.

Returns The private key as a string.

Return type str

static **readFile**(*fileName=None*, *mode='r'*)
Convenience method to read some data from a file.

Parameters

- **fileName** (*str*) – The file to read from, default = None
- **mode** (*str*) – The read mode, default = "r", Windows may require "rb"

Returns The data read from the file

Return type str

refresh()
refresh method.

Not relevant for CertSession but is included for consistency.

static **runCmd**(*cmd*)
Convenience method to run a command using subprocess.

Parameters **cmd** (*str*) – The command to run

Returns The output from the command

Return type str

Raises `subprocess.CalledProcessError` – If an non-zero return code is sent by the process

static **writeFile**(*fileName=None*, *mode='w'*, *fileData=None*)
Convenience method to write data to a file.

Parameters

- **fileName** (*str*) – The file to write to, default = None
- **mode** (*str*) – The write mode, default = "w"
- **fileData** (*varies*) – The data to write to the file

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

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.

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.

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

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

AbstractRequest

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

class cobra.mit.request.**AbstractRequest**
Abstract base class for all other request types.

options
str – The HTTP request query string for this object - readonly

id
None or int – An internal troubleshooting value useful for tracing the processing of a request within the cluster

uriBase
str – The base URI used to build the URL for queries and requests

__init__()
Instantiate an AbstractRequest instance.

getHeaders(*session*, *data=None*)
Get the headers for the session.

The data may be needed if a signature is needed to be calculated for a transaction.

Parameters

- **session** (cobra.mit.session.AbstractSession) – The session the headers should be for.
- **data** (*str, optional*) – The data for the request. The default is None

Returns A dictionary with the headers for the session.

Return type dict

getUriPathAndOptions (*session*)
Get the uri path and options.

Returns the full URI path and options portion of the URL that will be used in a query

Parameters **session** (cobra.mit.session.AbstractSession) – The session object which contains information needed to build the URI

Returns The URI and options strings

Return type str

id
Get the id.

Returns The id for this request.

Return type str

classmethod makeOptions (*options*)
Make the request options.

Returns a string containing the concatenated values of all key/value pairs for the options defined in dict options

Parameters **options** (*list*) – A list of options to turn into an option string

Returns The options strings

Return type str

options
Get the options.

Returns

All the options for this abstract request as a string joined by &'s.

Return type str

uriBase
Get the base uri.

Returns A string representing the base URI for this request.

Return type str

AbstractQuery

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

class cobra.mit.request.**AbstractQuery**
Abstract base class for a query.

options
str – The HTTP request query string for this object - readonly

propInclude
str – the current response property include filter. This filter can be used to specify the properties that should be included in the response. Valid values are:

•_all_

•naming-only

•config-explicit

- •config-all
- •config-only
- •oper

subtreePropFilter
str – The response subtree filter can be used to limit what is returned in a subtree response by property values

subtreeClassFilter
str – The response subtree class filter can be used to filter a subtree response down to one or more classes. Setting this can be done with either a list or a string, the value is always stored as a comma separated string.

subtreeInclude
str – The response subtree include filter can be used to limit the response to a specific type of information from the subtree, these include:

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

queryTarget
str – The query target filter can be used to specify what part of the MIT to query. You can query:

- •self - The object itself
- •children - The children of the object
- •subtree - All the objects lower in the heirarchy

classFilter
str – The target subtree class filter can be used to specify which subtree class to filter by. You can set this using a list or a string. The value is always stored as a comma separated string.

propFilter
str – The query target property filter can be used to limit which objects are returned based on the value that is set in the specific property within those objects.

subtree
str – The response subtree filter can be used to define what objects you want in the response. The possible values are:

- •no - No subtree requested
- •children - Only the children objects

•full - A full subtree

replica
int – The replica option can direct a query to a specific replica. The possible values are:

•1

•2

•3

orderBy
list or str – Request that the results be ordered in a certain way. This can be a list of property sort specifiers or a comma separated string. An example sort specifier: 'aaaUser.name|desc'.

pageSize
int – Request that the results that are returned are limited to a certain number, the pageSize.

id
None or int – An internal troubleshooting value useful for tracing the processing of a request within the cluster

uriBase
str – The base URI used to build the URL for queries and requests

__init__()
Instantiate an AbstractQuery instance.

classFilter
Get the class filter.

Returns The class filter (target-subtree-class)

Return type str

options
Get the options.

Returns

All the options for this abstract query as a string joined by &'s.

Return type str

orderBy
Get the orderBy sort specifiers string.

Returns The order-by string of sort specifiers.

Return type str

pageSize
Get the pageSize value.

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

Return type int

propFilter
Get the the property filter.

Returns The property filter (query-target-filter)

Return type str

propInclude
Get the property include.

Returns The property include (rsp-prop-include) value.

Return type str

queryTarget
Get the query target.

Returns The query target (query-target).

Return type str

replica
Get the replica.

Returns The replica option to be set on this query (replica).

Return type int

subtree
Get the subtree.

Returns The subtree specifier (rsp-subtree).

Return type str

subtreeClassFilter
Get the the subtree class filter.

Returns The subtree class filter (rsp-subtree-class)

Return type str

subtreeInclude
Get the subtree include.

Returns The subtree include (rsp-subtree-include) value.

Return type str

subtreePropFilter
Get the subtree property filter.

Returns The subtree property filter (rsp-subtree-filter) value.

Return type str

DnQuery

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

class cobra.mit.request.**DnQuery**(*dn*)
Query based on distinguished name (Dn).

options
str – The HTTP request query string string for this DnQuery object - readonly

dnStr
str – The base dn string for this DnQuery object - readonly

propInclude
str – the current response property include filter. This filter can be used to specify the properties that should be included in the response. Valid values are:

- _all_
- naming-only

- config-explicit
- config-all
- config-only
- oper

subtreePropFilter
str – The response subtree filter can be used to limit what is returned in a subtree response by property values

subtreeClassFilter
str – The response subtree class filter can be used to filter a subtree response down to one or more classes. Setting this can be done with either a list or a string, the value is always stored as a comma separated string.

subtreeInclude
str – The response subtree include filter can be used to limit the response to a specific type of information from the subtree, these include:

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

queryTarget
str – The query target filter can be used to specify what part of the MIT to query. You can query:

- self - The object itself
- children - The children of the object
- subtree - All the objects lower in the heirarchy

classFilter
str – The target subtree class filter can be used to specify which subtree class to filter by. You can set this using a list or a string. The value is always stored as a comma separated string.

propFilter
str – The query target property filter can be used to limit which objects are returned based on the value that is set in the specific property within those objects.

subtree
str – The response subtree filter can be used to define what objects you want in the response. The possible values are:

- no - No subtree requested

- children - Only the children objects
- full - A full subtree

orderBy
list or str – Request that the results be ordered in a certain way. This can be a list of property sort specifiers or a comma separated string. An example sort specifier: 'aaaUser.nameldesc'.

pageSize
int – Request that the results that are returned are limited to a certain number, the pageSize.

replica
int – The replica option can direct a query to a specific replica. The possible values are:

- 1
- 2
- 3

id
None or int – An internal troubleshooting value useful for tracing the processing of a request within the cluster

uriBase
str – The base URI used to build the URL for queries and requests

__init__(*dn*)
Initialize a DnQuery object.

Parameters **dn** (*str or* cobra.mit.naming.Dn) – The Dn to query

dnStr
Get the dn string.

Returns The dn string for this dn query.

Return type str

getUrl(*session*)
Get the URL containing all the query options.

Parameters **session** (cobra.mit.session.AbstractSession) – The session to use for this query.

Returns The url

Return type str

options
Get the options.

Returns

All the options for this dn queryas a string joined by &'s.

Return type str

ClassQuery

Class that creates a query object based on object class.

class cobra.mit.request.**ClassQuery**(*className*)
Query based on class name.

options
str – The HTTP request query string string for this DnQuery object - readonly

className
str – The className to query for - readonly

propInclude
str – the current response property include filter. This filter can be used to specify the properties that should be included in the response. Valid values are:

•_all_

•naming-only

•config-explicit

•config-all

•config-only

•oper

subtreePropFilter
str – The response subtree filter can be used to limit what is returned in a subtree response by property values

subtreeClassFilter
str – The response subtree class filter can be used to filter a subtree response down to one or more classes. Setting this can be done with either a list or a string, the value is always stored as a comma separated string.

subtreeInclude
str – The response subtree include filter can be used to limit the response to a specific type of information from the subtree, these include:

•audit-logs

•event-logs

•faults

•fault-records

•health

•health-records

•relations

•stats

•tasks

•count

•no-scoped

•required

queryTarget
str – The query target filter can be used to specify what part of the MIT to query. You can query:

•self - The object itself

•children - The children of the object

•subtree - All the objects lower in the heirarchy

classFilter
str – The target subtree class filter can be used to specify which subtree class to filter by. You can set this using a list or a string. The value is always stored as a comma separated string.

propFilter
str – The query target property filter can be used to limit which objects are returned based on the value that is set in the specific property within those objects.

subtree
str – The response subtree filter can be used to define what objects you want in the response. The possible values are:

- no - No subtree requested
- children - Only the children objects
- full - A full subtree

orderBy
list or str – Request that the results be ordered in a certain way. This can be a list of property sort specifiers or a comma separated string. An example sort specifier: 'aaaUser.nameldesc'.

pageSize
int – Request that the results that are returned are limited to a certain number, the pageSize.

replica
int – The replica option can direct a query to a specific replica. The possible values are:

- 1
- 2
- 3

id
None or int – An internal troubleshooting value useful for tracing the processing of a request within the cluster

uriBase
str – The base URI used to build the URL for queries and requests

__init__(*className*)
Initialize a ClassQuery instance.

Parameters **className** (*str*) – The className to query for

className
Get the class name.

Returns The class name for this class query

Return type str

getUrl(*session*)
Get the URL containing all the query options.

Parameters **session** (cobra.mit.session.AbstractSession) – The session to use for this query.

Returns The url

Return type str

options
Get the options.

Returns

All the options for this class query as a string joined by &'s.

Return type str

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**

Change the configuration.

cobra.mit.access.MoDirectory.commit() function uses this class.

options

str – The HTTP request query string string for this DnQuery object - readonly

data

str – The payload for this request in JSON format - readonly

xmldata

str – The payload for this request in XML format - readonly

subtree

str – The response subtree filter can be used to define what objects you want in the response. The possible values are:

- no - No subtree requested
- children - Only the children objects
- full - A full subtree

id

None or int – An internal troubleshooting value useful for tracing the processing of a request within the cluster

uriBase

str – The base URI used to build the URL for queries and requests

__init__()

Initialize a ConfigRequest instance.

addMo(*mo*)

Add a managed object (MO) to the configuration request.

Args mo (cobra.mit.mo.Mo): The managed object to add

Raises ValueError – If the context root of the MO is not allowed. This can happen if the MO being added does not have a common context root with the MOs that are already added to the configuration request

data

Get the data as JSON.

Raises `CommitError` – If no Mo's have been added to this config request.

Returns The data that will be committed as a JSON string.

Return type str

getRootMo()
Get the Root Mo for this configuration request.

Returns The root Mo for the config request

Return type None or *cobra.mit.mo.Mo*

getUriPathAndOptions(*session*)
Get the full URI path and options portion of the URL.

Parameters **session** (`cobra.mit.session.AbstractSession`) – The session object which contains information needed to build the URI

Returns The URI and options string

Return type str

getUrl(*session*)
Get the URL containing all the query options.

Parameters **session** (`cobra.mit.session.AbstractSession`) – The session to use for this query.

Returns The url

Return type str

hasMo(*dn*)
Check if the configuration request has a specific MO.

Args: dn (str): The distinguished name of the mo to check

Returns (bool): True if the MO is in the configuration request, otherwise False

options
Get the options.

Returns

All the options for this config request as a string joined by &'s.

Return type str

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

Parameters **mo** (`cobra.mit.mo.Mo`) – The managed object to add

requestargs(*session*)
Get the arguments to be used by the HTTP request.

session (cobra.mit.session.AbstractSession): The session to be used to build the the request arguments

Returns The arguments

Return type dict

subtree
Get the subtree.

Returns The subtree specifier.

Return type str

xmldata
Get the data as XML.

Raises CommitError – If no Mo's ahve been added to this config request.

Returns The data as a XML string.

Return type str

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*)

Hybrid query and request for tags.

This class does both setting of tags (request) and retrieving of tags (query).

options

str – The HTTP request query string string for this DnQuery object - readonly

data

str – The payload for this request in JSON format - readonly

dnStr

str – The base Dn for this request/query - readonly

add

None or str or list – The tag(s) to add, default is None

remove

None or str or list – The tag(s) to remove, default is None

id

None or int – An internal troubleshooting value useful for tracing the processing of a request within the cluster

uriBase

str – The base URI used to build the URL for queries and requests

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

Initialize a tags query/request object.

Parameters

- **dn** (*str or cobra.mit.naming.Dn*) – The base Dn for this request/query
- **add** (*None or str or list*) – The tag(s) to add, default is None
- **remove** (*None or str or list*) – The tag(s) to remove, default is None

add

Get the add string.

Returns The string of tags that will be added by this request.

Return type str

data

Get the data.

Currently only JSON is supported

Returns The data that will be committed as a JSON string.

Return type str

dnStr

Get the dn string.

Returns

The string representing the Dn that the tags will be committed to.

Return type str

getUrl (*session*)
Get the URL containing all the query options.

Parameters **session** (cobra.mit.session.AbstractSession) – The session to use for this query.

Returns The url

Return type str

options
Get the options.

Returns

All the options for this tags request as a string joined by &'s.

Return type str

remove
Get the remove string.

Returns The string of tags that will be removed by this request.

Return type str

requestargs (*session*)
Get the arguments to be used by the HTTP request.

session (cobra.mit.session.AbstractSession): The session to be used to build the the request arguments

Returns The arguments

Return type dict

TraceQuery

A class that creates a trace query

class cobra.mit.request.**TraceQuery** (*dn*, *targetClass*)
Trace Query using a base Dn and a target class.

options
str – The HTTP request query string string for this DnQuery object - readonly

targetClass
str – The targetClass for this trace query

dnStr
str – The base Dn string for this trace query

propInclude
str – the current response property include filter. This filter can be used to specify the properties that should be included in the response. Valid values are:

•_all_

•naming-only

•config-explicit

•config-all

•config-only

- oper

subtreePropFilter
str – The response subtree filter can be used to limit what is returned in a subtree response by property values

subtreeClassFilter
str – The response subtree class filter can be used to filter a subtree response down to one or more classes. Setting this can be done with either a list or a string, the value is always stored as a comma separated string.

subtreeInclude
str – The response subtree include filter can be used to limit the response to a specific type of information from the subtree, these include:

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

queryTarget
str – The query target filter can be used to specify what part of the MIT to query. You can query:

- self - The object itself
- children - The children of the object
- subtree - All the objects lower in the heirarchy

classFilter
str – The target subtree class filter can be used to specify which subtree class to filter by. You can set this using a list or a string. The value is always stored as a comma separated string.

propFilter
str – The query target property filter can be used to limit which objects are returned based on the value that is set in the specific property within those objects.

subtree
str – The response subtree filter can be used to define what objects you want in the response. The possible values are:

- no - No subtree requested
- children - Only the children objects
- full - A full subtree

orderBy
list or str – Request that the results be ordered in a certain way. This can be a list of property sort specifiers or a comma separated string. An example sort specifier: 'aaaUser.nameldesc'.

pageSize
int – Request that the results that are returned are limited to a certain number, the pageSize.

replica
int – The replica option can direct a query to a specific replica. The possible values are:

- 1
- 2
- 3

id
None or int – An internal troubleshooting value useful for tracing the processing of a request within the cluster

uriBase
str – The base URI used to build the URL for queries and requests

__init__(*dn*, *targetClass*)
Initialize a TraceQuery instance.

Parameters

- **dn** (*str or cobra.mit.naming.Dn*) – The base Dn for this query
- **targetClass** (*str*) – The target class for this query

dnStr
Get the base dn string.

Returns The string representing the base Dn for this trace query.

Return type str

getUrl(*session*)
Get the URL containing all the query options.

Parameters **session** (cobra.mit.session.AbstractSession) – The session to use for this query.

Returns The url

Return type str

options
Get the options.

Returns

All the options for this trace query as a string joined by &'s.

Return type str

targetClass
Get the target class.

Returns The string representing the target class for this trace query.

Return type str

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.

UploadPackage

Class for uploading L4-L7 device packages to APIC

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

data
str – A string containing the payload for this request in JSON format - readonly

devicePackagePath
str – Path to the device package on the local file system. No Path verification is performed, so any errors accessing the specified file will be raised directly to the calling function.

Note: If validation is requested, the device package contents are verified to contain a device specification XML/JSON document

---

options
str – The HTTP request query string for this object - readonly

id
None or int – An internal troubleshooting value useful for tracing the processing of a request within the cluster

uriBase
str – The base URI used to build the URL for queries and requests

__init__(*devicePackagePath*, *validate=False*)
Upload a device package to an APIC.

cobra.mit.access.MoDirectory.commit() is required to commit the upload.

Parameters

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

data
Get the data for the request.

devicePackagePath
Get the device package path.

Returns The path to the device package.

Return type str

getUrl (*session*)
Get the URL for this request, includes all options as well.

Parameters **session** (cobra.mit.session.AbstractSession) – The session to use for this query.

Returns A string containing the request url

Return type str

requestargs (*session*)
Get the request arguments for this object.

Parameters **session** (cobra.mit.session.AbstractSession) – The session to be used to build the the requestarguments

Returns A dictionary containing the arguments

Return type dict

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.

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*)
Creates a connection to the APIC and the MIT.

MoDirectory requires an existing session.

__init__ (*session*)
Initialize a MoDirectory instance.

Parameters **session** (cobra.mit.session.AbstractSession) – The session

commit (*configObject*)
Commit operation for a request object.

Commit a change on the APIC or fabric node.

Parameters **configObject** (cobra.mit.request.AbstractRequest) – The configuration request to commit

Returns The response as a string

Return type str

Raises CommitError – If no MOs have been added to the config request

login()
Create a session to an APIC.

logout()
End a session to an APIC.

lookupByClass(*classNames*, *parentDn=None*, ***kwargs*)
Lookup MO's by class.

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

Parameters

- **classNames** (*str or list*) – The class name list of class names. If parentDn is set, the classNames are used as a filter in a subtree query for the parentDn
- **parentDn** (*cobra.mit.naming.Dn or str, optional*) – The distinguished name of the parent object as a *cobra.mit.naming.Dn* or string.
- ****kwargs** – Arbitrary parameters to be passed to the query generated internally, to further filter the result

Returns A list of the managed objects found in the query.

Return type list

lookupByDn(*dnStrOrDn*, ***kwargs*)
Query the APIC or fabric node by distinguished name (Dn).

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

Parameters

- **dnStrOrDn** (*str or cobra.mit.naming.Dn*) – A distinguished name as a *cobra.mit.naming.Dn* or string
- ****kwargs** – Arbitrary parameters to be passed to the query generated internally, to further filter the result

Returns

None if no MO was returned otherwise *cobra.mit.mo.Mo*

Return type None or *cobra.mit.mo.Mo*

query(*queryObject*)
Query the Model Information Tree.

The various types of potential queryObjects provide a variety of search options

Parameters **queryObject** (*cobra.mit.request.AbstractRequest*) – A query object

Returns A list of Managed Objects (MOs) returned from the query

Return type list

reauth()
Re-authenticate the 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.

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.

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'
```

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.

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.

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*)

Represents managed objects (MOs).

Managed objects (MOs) represent a physical or logical entity with a set of configurations and properties.

dn

cobra.mit.naming.Dn – The distinguished name (Dn) of the managed object (MO) - readonly

rn

cobra.mit.naming.Rn – The relative name (Rn) of the managed object (MO) - readonly

status
cobra.internal.base.moimpl.MoStatus – The status of the MO - readonly

parentDn
cobra.mit.naming.Dn – The parent managed object (MO) distinguished name (Dn) - readonly

parent
cobra.mit.mo.Mo – The parent managed object (MO) - readonly

dirtyProps
set – modified properties that have not been committed - readonly

children
cobra.internal.base.moimpl.BaseMo._ChildContainer – A container for the children of this managed object - readonly

numChildren
int – The number of direct decendents for this managed object - readonly

contextRoot
None or cobra.mit.mo.Mo – The managed object that is the context root for this managed object

__getattr__(*propName*)
Implement getattr().

__init__(*parentMoOrDn*, *markDirty*, **namingVals*, ***creationProps*)
Initialize a managed object (MO).

This should not be called directly. Instead initialize the Mo from the model that you need.

Parameters

- **parentMoOrDn** (*str or cobra.mit.naming.Dn or cobra.mit.mo.Mo*) – The parent managed object (MO) or distinguished name (Dn).
- **markDirty** (*bool*) – If True, the MO is marked has having changes that need to be committed. If False the Mo is not marked as having changes that need to be committed.
- ***namingVals** – Required values that are used to name the Mo, i.e. they become part of the MOs distinguished name.
- ****creationProps** – Properties to be set at the time the MO is created, these properties can also be set after the property is created if needed.

Raises `NotImplementedError` – If this class is called directly

__setattr__(*propName*, *propValue*)
Implement setattr().

children
Get the children iterator.

Returns An iterator for the children of this Mo.

Return type iterator

contextRoot
Get the context root of the distinguished name.

Returns

If the Dn has no context root. cobra.mit.mo.Mo: The managed object that is the context root for

this managed object if the Dn has a context root.

Return type None

delete()
Mark the Mo ad deleted.

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

dirtyProps
Get the properties that are marked as dirty.

Returns The set of properties that are dirty.

Return type set

dn
Get the distinguished name.

Returns The Dn for this Mo.

Return type *cobra.mit.naming.Dn*

isPropDirty(*propName*)
Check if a property has been modified on this managed object.

Parameters **propName** (*str*) – The property name as a string

Returns

True if the property has been modified and not commited, False otherwise

Return type bool

numChildren
Get the number of children.

Returns The number of children that this Mo has.

Return type int

parent
Get the parent Mo.

Returns The parent Mo.

Return type *cobra.mit.mo.Mo*

parentDn
Get the parent distinguished name.

Returns The parent Dn.

Return type *cobra.mit.naming.Dn*

resetProps()
Reset the managed object (MO) properties.

This will discard uncommitted changes.

rn
Get the relative name.

Returns The relative name for this Mo.

Return type *cobra.mit.naming.Rn*

status
Get the status.

Returns The status for this Mo.

Return type cobra.internal.base.moimpl.MoStatus

Meta Module

The following sections describe the classes in the meta module.

Category

Class that represents an object category.

class cobra.mit.meta.**Category**(*name*, *categoryId*)
Category class for Managed Object (MO) class meta or property meta.

Used to classify MOs or MO properties into various categories. The categories are defined in the ACI model package for ever MO property.

__eq__(*other*)
Implement ==.

__ge__(*other*)
Implement >=.

__gt__(*other*)
Implement >.

__init__(*name*, *categoryId*)
Initialize a MO property category.

Parameters

- **name** (*str*) – The name of the category
- **categoryId** (*int*) – The integer representing the category id

__le__(*other*)
Implement <=.

__lt__(*other*)
Implement <.

__ne__(*other*)
Implement !=.

__str__()
Implement str().

ClassLoader

Class that loads a specified class.

class cobra.mit.meta.**ClassLoader**
Import a class by name.

A convenience class to import classes from a string containing the class name

classmethod loadClass(*fqClassName*)
Load a class from a fully qualified name.

Parameters **fqClassName** (*str*) – A fully qualified class name as in package.module.class. For example: cobra.model.pol.Uni

Returns The imported class

Return type *cobra.mit.mo.Mo*

ClassMeta

Class that provides information about an object class.

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

Represents a classes metadata.

className
str – The class name for the meta

moClassName
None or str – The class name for the MO

label
str – The label for the class meta

category
None or cobra.mit.meta.Category – The class category

isAbstract
bool – True if the class is abstract, False otherwise

isRelation
bool – True if the class is a relationship object, False otherwise

isSource
bool – True if the class is a source relationship object, False otherwise

isExplicit
bool – True if the object is an explicit relationship, False if the object forms an indirect named relationship

isNamed
bool – True if the object is a named source relationship object, False otherwise

writeAccessMask
long – The write permissions for this class

readAccessMask
long – The read permissions for this class

isDomainable
bool – True if the MO is domainable, False otherwise

isReadOnly
bool – True if the MO is readonly, False otherwise

isConfigurable
bool – True if the MO can be configured, False otherwise

isDeletable
bool – True if the MO can be deleted

isContextRoot
bool – True if the MO is the context root

concreteSubClasses
cobra.mit.meta.ClassMeta._ClassContainer – A container that keeps track of all the subclasses that are concrete

superClasses
cobra.mit.meta.ClassMeta._ClassContainer – A container that keeps track of all the super classes

childClasses
cobra.mit.meta.ClassMeta._ClassContainer – A container that keeps track of the actual child classes

childNamesAndRnPrefix
list of tuples – A list containing tuples where the first element is the child name and the second element is the rn prefix

parentClasses
cobra.mit.meta.ClassMeta._ClassContainer – A container that keeps track of the actual parent classes

props
cobra.mit.meta._PropContainer – A container that keeps track of all of the classes properties

namingProps
list – A list containing `cobra.mit.meta.PropMeta` for each property that is a naming property.

rnFormat
None or str – A string representing the relative name format

rnPrefixes
list of tuples – The relative name prefixes where the first element in the tuple is the rn prefix and the second element is a bool where True means the prefix has naming properties and False otherwise.

ctxRoot
None or cobra.mit.mo.Mo – The context root for this class.

__init__(*className*)
Initialize a ClassMeta instance.

Parameters **className** (*str*) – The class name for this meta object

getClass()
Use the className to import the class for this meta object.

Returns The imported class for this meta object

Return type mixed

getContextRoot(*pStack=None*)
Get the meta's context root.

Parameters **pStack** (*set*) – The parent stack

Returns The class of the context root

Return type None or *cobra.mit.mo.Mo*

hasContextRoot()
Check if the meta has a context root.

Returns True if the meta has a context root and False otherwise

Return type boo

Constant

class cobra.mit.meta.**Constant**(*const*, *label*, *value*)
A class to represent constants for properties.

__eq__(*other*)
Implement ==.

__ge__(*other*)
Implement >=.

__gt__(*other*)
Implement >.

__init__(*const*, *label*, *value*)
Initialize a constant object.

Parameters

- **const** (*str*) – The constant string that can be used for the property
- **label** (*str*) – The label for this constant
- **value** (*int*) – The value for this constant

__le__(*other*)
Implement <=.

__lt__(*other*)
Implement <.

__ne__(*other*)
Implement !=.

__str__()
Implement str().

NamedSourceRelationMeta

class cobra.mit.meta.**NamedSourceRelationMeta**(*className*, *targetClassName*)
The meta data for a named source relationship object.

__init__(*className*, *targetClassName*)
Initialize a named source relationship meta object.

Parameters

- **className** (*str*) – The source Mo class name for the relationship
- **targetClassName** (*str*) – The target class name for the relationship

PropMeta

class cobra.mit.meta.**PropMeta**(*typeClassName*, *name*, *moPropName*, *propId*, *category*)
The meta data for properties of managed objects.

typeClass
str – The class of the property

name
str – The name of the property

moPropName
str – The managed object property name

id
None or int – The property id

category
cobra.mit.meta.Category – The property category object

help
None or str – The help string for the property

label
None or str – The label for the property

unit
None or str – The units the property is in

defaultValue
None or str – The default value for the property

isDn
bool – True if the property is a distingushed name, False otherwise

isRn
bool – True if the property is a relative name, False otherwise

isConfig
bool – True if the property is a configuration property, False otherwise

isImplicit
bool – True if the property is implicitly defined, False otherwise

isOper
bool – True if the property is an operations property, False otherwise

isAdmin
bool – True if the property is an admin property, False otherwise

isCreateOnly
bool – True if the property can only be set when the MO is created, False otherwise

isNaming
bool – True if the property is a naming property, False otherwise

isStats
bool – True if the property is a stats property, False otherwise

isPassword
bool – True if the property is a password property, False otherwise

needDelimiter
bool – True if the property needs delimiters, False otherwise

constants
dict of cobra.mit.meta.Constants – A dictionary where the keys are the constants const and the values are the constants objects

constsToLabels
dict – A dictionary mapping the properties constants consts to the constants label

labelsToConsts
dict – A dictionary mapping the properties constants labels to the constants consts

__eq__(*other*)
Implement ==.

__ge__(*other*)
Implement >=.

__gt__(*other*)
Implement >.

__hash__()
Implement hash().

__init__(*typeClassName*, *name*, *moPropName*, *propId*, *category*)
Initialize a PropMeta instance.

Parameters

- **typeClassName** (*str*) – The class for the type of python object that should be used to represent this property
- **moPropName** (*str*) – The managed object property name
- **propId** (*int*) – The property Id number
- **category** (cobra.mit.meta.Category) – The property category

__le__(*other*)
Implement <=.

__lt__(*other*)
Implement <.

__ne__(*other*)
Implement !=.

__str__()
Implement str().

isValidValue(*value*)
Check a value against the validators in the meta.

Parameters **value** (*str*) – The value to check

Returns True if the value is valid for this property or False otherwise

Return type bool

static makeValue(*value*)
Create a property using a value.

Parameters **value** (*str*) – The value to set the property to

Returns The value

Return type str

SourceRelationMeta

class cobra.mit.meta.**SourceRelationMeta**(*className*, *targetClassName*)
The meta data for a source object in a relationship.

__init__(*className*, *targetClassName*)
Initialize a source relationship meta object.

Parameters

- **className** (*str*) – The source Mo class name for the relationship
- **targetClassName** (*str*) – The target class name for the relationship

getTargetClass()
Import and returns the target class for a relationship.

Returns The target class

Return type *cobra.mit.mo.Mo*

TargetRelationMeta

class cobra.mit.meta.**TargetRelationMeta**(*className*, *sourceClassName*)
The meta data for a target object in a relationship.

__init__(*className*, *sourceClassName*)
Initialize a target relationship meta object.

Parameters

- **className** (*str*) – The target Mo class name for the relationship
- **sourceClassName** (*str*) – The source class name for the relationship

getSourceClass()
Import and return the source class.

Returns The source class

Return type *cobra.mit.mo.Mo*

CHAPTER 8

Examples

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.

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.

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.

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')
```

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')
```

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')

# 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)
```

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)
```

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',
                        name='encap')
nsCfg = ConfigRequest()
```

```
nsCfg.addMo(fvnsVlanInstP)
moDir.commit(nsCfg)
```

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)
```

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.

```
from __future__ import print_function
# Copyright 2015 Cisco Systems, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

#!/usr/bin/env python


# Import access classes
from cobra.mit.access import MoDirectory
from cobra.mit.session import LoginSession
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...')

    vmmpVMwareProvPMo = moDir.lookupByDn('uni/vmmp-VMware')
    vmmDomPMo = DomP(vmmpVMwareProvPMo, VMM_DOMAIN_INFO['name'])

    vmmUsrMo = []
```

```
    for usrp in VMM_DOMAIN_INFO['usrs']:
        usrMo = UsrAccP(vmmDomPMo, usrp['name'], usr=usrp['usr'],
                        pwd=usrp['pwd'])
        vmmUsrMo.append(usrMo)

    # Create Controllers under domain
    for ctrlr in VMM_DOMAIN_INFO['ctrlrs']:
        vmmCtrlrMo = CtrlrP(vmmDomPMo, ctrlr['name'], scope=ctrlr['scope'],
                            hostOrIp=ctrlr['ip'])
        # Associate Ctrlr to UserP
        RsAcc(vmmCtrlrMo, tDn=vmmUsrMo[0].dn)

    # Associate Domain to Namespace
    RsVlanNs(vmmDomPMo, tDn=fvnsVlanInstPMo.dn)

    vmmCfg = ConfigRequest()
    vmmCfg.addMo(vmmDomPMo)
    moDir.commit(vmmCfg)
    print("VMM Domain Creation Completed.")

    print("Starting Tenant Creation..")
    for tenant in TENANT_INFO:
        print("Creating tenant %s.." % (tenant['name']))
        fvTenantMo = Tenant(uniMo, tenant['name'])

        # Create Private Network
        Ctx(fvTenantMo, tenant['pvn'])

        # Create Bridge Domain
        fvBDMo = BD(fvTenantMo, name=tenant['bd'])

        # Create association to private network
        RsCtx(fvBDMo, tnFvCtxName=tenant['pvn'])

        # Create Application Profile
        for app in tenant['ap']:
            print('Creating Application Profile: %s' % app['name'])
            fvApMo = Ap(fvTenantMo, app['name'])

            # Create EPGs
            for epg in app['epgs']:

                print("Creating EPG: %s..." % (epg['name']))
                fvAEPgMo = AEPg(fvApMo, epg['name'])

                # Associate EPG to Bridge Domain
                RsBd(fvAEPgMo, tnFvBDName=tenant['bd'])
                # Associate EPG to VMM Domain
                RsDomAtt(fvAEPgMo, vmmDomPMo.dn)

        # Commit each tenant seperately
        tenantCfg = ConfigRequest()
        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)

```

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.

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']
```

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.

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.

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*.

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*.

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*.

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.

Authentication Error

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

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.

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)
```

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
```

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.

Python Version Error

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

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*

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*

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

B

C

D

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X