AOS PyEZ

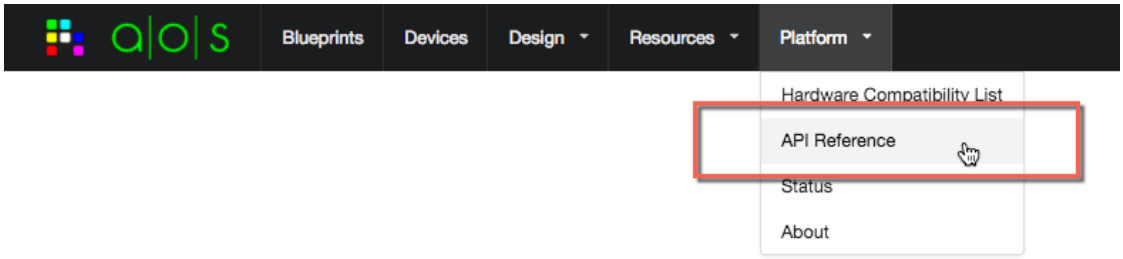
Release 0.6.0

Aug 31, 2017

Contents

Release version 0.6. (*Changelog*)

The aos-pyez library is a Pythonic interface to the Apstra AOS-Server API. The complete AOS-Server API documentation is available directly from the AOS-Server User Interface, as shown:



The aos-pyez library is designed to work with AOS version 1.1. At present, the aos-pyez library exposes some, but not all functionality provided by the AOS-Server. The aos-pyez library, does however, provide you the means to access any aspect of the API, as described in the *Session.api.requests* guide. With the current aos-pyez, you will be able to use the Design, Resource, and Blueprint Build features. Additional features will be continuously added. If you'd like to get involved, please post your requests/bugs on the github repo.

CHAPTER 1

Guide

Installation

To get the latest release, run

```
$ pip install aos-pyez
```

To get latest development version of aos-pyez, run

```
$ pip install git+https://github.com/Apstra/aos-pyez.git
```

Quickstart

This section will guide you through the process of creating a management session with the AOS-Server and performing a few basic interactions.

Let's start with creating a session:

```
>>> from apstra.aosom.session import Session
>>> aos = Session('192.168.59.250', user='admin', passwd='admin')
>>> aos.login()
```

The first parameter is the AOS-server IP address or hostname. The remaining key/value arguments are documented in the API reference section. The `user` and `passwd` both default to *admin* if not provided.

The `login()` method will make the request to authenticate a login and provide back a session token. If for any reason the login attempt fails, an exception will be raised. See *`Session.login()`* for details. You can verify the login session information by examinging the *`Session.session`*.

The value of the `aos.session` value looks like:

```
>>> aos.session
{'port': 8888,
 'server': 'aos-server',
 'token': u'eyJhbGci<~snip~>MTUiMP0skQ'}
```

The *Session* then allows you to access other API features. These features are generally a collection of similar items. These features are defined within the `Session.ModuleCatalog`.

```
['Blueprints', 'IpPools', 'DesignTemplates', 'ExternalRouters',
 'AsnPools', 'RackTypes', 'LogicalDevices', 'Devices', 'LogicalDeviceMaps']
```

- The `Devices` feature allows you to access the Device-Manager features; i.e. access inventory management and information about the devices being managed by AOS.
- The `AsnPools`, `IpPools`, and `ExternalRouters` are resources that you provide to AOS so that they can be assigned and used to services that you define within AOS - aka "Blueprints".
- The `DesignTemplates`, `LogicalDevices`, `LogicalDevicesMaps`, and `RackTypes` are all design elements. You use these design elements to define your Blueprint services.
- Finally the `Blueprints` are the network services that you are managing with AOS.

To access any of these, simply use the name as an attribute of the aos Session. By way of example, here is how you can access the devices under management, and display the Management-IpAddr, Serial-Number, Model as it is known to AOS, and the OS/version information:

```
>>> for dev in aos.Devices:
...     dev_facts = dev.value['facts']
...     print dev_facts['mgmt_ipaddr'], dev_facts['serial_number'], dev_facts['aos_hcl_
↪model'], dev_facts['os_version']
192.168.60.16 08002737C2C1 Cumulus_VX 3.1.1
192.168.60.19 080027A71AE6 Arista_vEOS 4.16.6M
192.168.60.18 0800277025C8 Cumulus_VX 3.1.1
192.168.60.17 080027AC6320 Cumulus_VX 3.1.1
192.168.60.15 08002763CBDC Cumulus_VX 3.1.1
```

Client Session

This guide explains how you can make a connection to the AOS-server and using the elements of the Session.

Creating a Session

The first step is creating a client Session to the AOS-Server:

```
>>> from apstra.aosom.session import Session
>>> aos = Session('192.168.59.250', user='admin', passwd='admin)
>>> aos.login()
```

The first parameter is the AOS-server IP address or hostname. The remaining key/value arguments are documented in the API reference section. The user and passwd both default to `admin` if not provided.

The `login()` method will make the request to authenticate a login and provide back a session token. If for any reason the login attempt fails, an exception will be raised. See *session.Session.login()* for details. You can verify the login session information by examining the `Session.session`.

The value of the `aos.session` value looks like:

```
>>> aos.session
{'port': 8888,
 'server': 'aos-server',
 'token': u'eyJhbGci<~snip~>MTUiMP0skQ'}
```

Making use of a Session

The general use of the Session an interface to the exposed API features, such as the Device-Manager, using resources, design elements, and managing network services. You can see a list of the existing API features exposed via the aos-pyez library by examining the `Session.ModuleCatalog` data:

```
['Blueprints', 'IpPools', 'DesignTemplates', 'ExternalRouters',
 'AsnPools', 'RackTypes', 'LogicalDevices', 'Devices', 'LogicalDeviceMaps']
```

To access any of these features, you simple use the name of the module as an attribute of the Session. For example, to show a list of IP Pool names managed by AOS, you would do the following:

```
>>> aos.IpPools.names
[u'Switches-IpAddrs', u'Servers-IpAddrs']
```

The modules are generally managed as a `collection.Collection` and you can manage an item within a collection as a `collection.CollectionItem`. The use of Collections and CollectionItems will be covered on separate guide pages.

Resuming an Existing Session

In some cases, you may want to *pass around* the session information between programs. Do do this you can use the `Session.session` property. For example returning the session as JSON data:

Listing 1.1: save-session.py

```
>>> keep_session = aos.session
>>> json.dump(keep_session, open('keep_session.json', 'w+'), indent=2)
```

And then in a different program, you can use this session data to restore a connection:

Listing 1.2: restore-session.py

```
>>> aos = Session()
>>> had_session = json.load(open('keep_session.json'))
>>> aos.session = had_session
```

And now the session is again active. If the session could not be restored for any reason, then an exceptions will be raised. See the `Session.session.login()` for details on those exceptions.

Session.api.requests

The Session instance maintains properties that allows you direct Requests level access, should you need it for any reason. For example, you may want access to API capabilities not presently exposed by the aos-pyez library. You can use the `Session.api` and `Session.api.requests` values. The `api.url` maintains the top level HTTP URL to the AOS-Server:

```
>>> aos.api.url
'http://aos-server:8888/api'
```

And the `api.requests` is a Requests Session object used for direct access. Here is an example of directly invoking a GET on API version build information:

```
>>> aos.api.requests.get("%s/versions/build" % aos.api.url)
<Response [200]>
```

And the data returned by using the `api.requests` is the same Requests Response object:

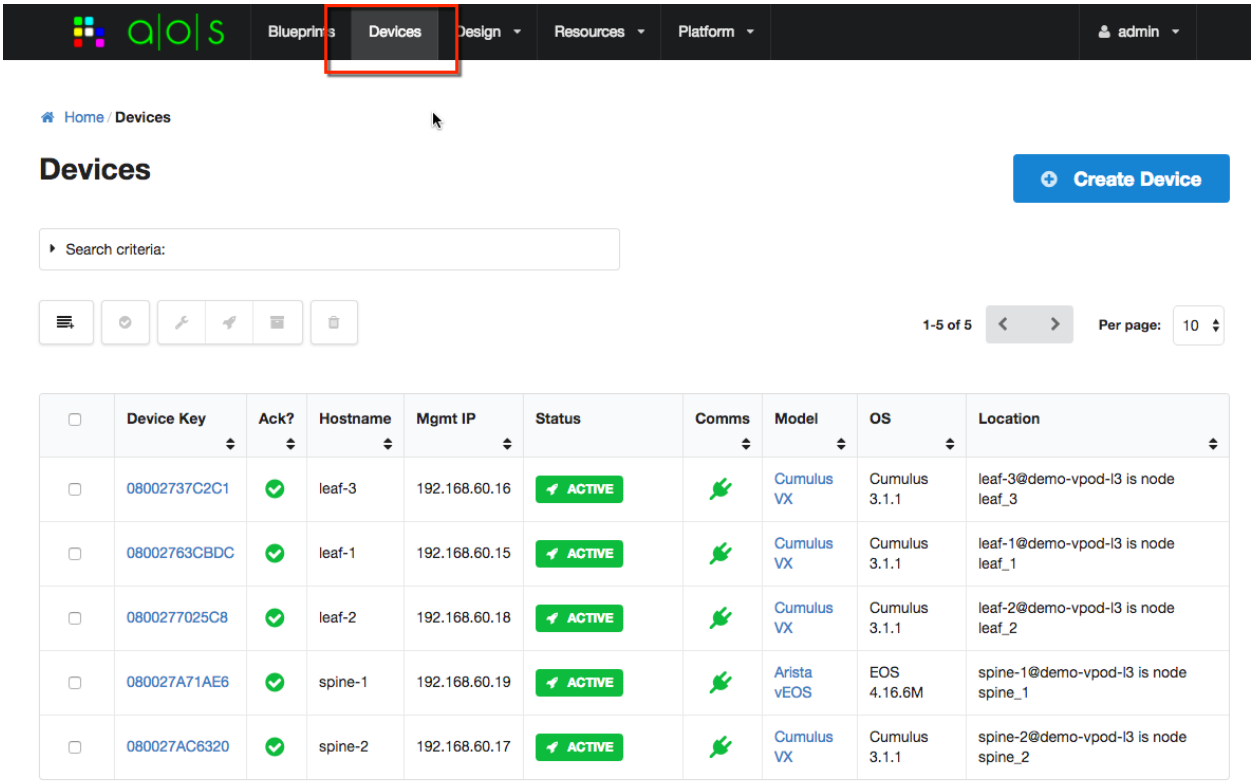
```
>>> got = aos.api.requests.get("%s/versions/build" % aos.api.url)
>>> got.json()
{u'version': u'1.1.0-11', u'build_datetime': u'2016-12-12_16:46:51_PST'}
```

The `Session.api.requests` property stores the necessary headers from the login authentication. This means that you do not need to explicitly provide the `headers=` value on the requests call. For example, doing a GET on the IP Pools would require an authentication token. Here is how you could directly invoke the requests library to do the same data retrieval as shown before, in this case getting the raw JSON output.

```
# each collection has a url property as well!
>>> aos.IpPools.url
'http://aos-server:8888/api/resources/ip-pools'

>>> got = aos.api.requests.get(aos.IpPools.url)
>>> print json.dumps(got.json(), indent=2)
{
  "items": [
    {
      "status": "in_use",
      "subnets": [
        {
          "status": "pool_element_in_use",
          "network": "172.20.0.0/16"
        }
      ],
      "display_name": "Switches-IpAddrs",
      "tags": [],
      "created_at": "2017-01-28T19:57:12.887618Z",
      "last_modified_at": "2017-01-28T19:57:12.887618Z",
      "id": "65dfbc77-1c77-4a99-98a6-e36c5aa7e4d0"
    },
    {
      "status": "in_use",
      "subnets": [
        {
          "status": "pool_element_in_use",
          "network": "172.21.0.0/16"
        }
      ],
      "display_name": "Servers-IpAddrs",
      "tags": [],
      "created_at": "2017-01-28T19:57:13.096657Z",
      "last_modified_at": "2017-01-28T19:57:13.096657Z",
      "id": "0310d821-d075-4075-bdda-55cc6df57258"
    }
  ]
}
```

Devices

You can see all devices under AOS management from the Devices UI:



For this version of aos-pyez you are able to manage devices as a collection, as described in the *Collections* guide pages. You can access the devices by using the `Devices` property of the

Session:

```
>>> aos.Devices
```

You can select a specific device from the collection by using the *device-key* as shown in the UI. The device-key is generally the serial-number of the device.

If you are using the python interactive interpreter, you can see a list of all known device-keys, for example:

```
>>> aos.Devices
{
   "url": "systems",
   "by_id": "id",
   "item-names": [
      "08002737C2C1",
      "080027A71AE6",
      "0800277025C8",
      "080027AC6320",
      "08002763CBDC"
   ],
```

```
   "by_label": "device_key"
}
```

And in the above, you can see that the `label` used to index the collection is the `device_key` collection item property.

Available Device Information

AOS collects information about each device under management. Some of this information is speciifc to AOS, for example the version of the AOS device agent. Some of this information is specific to the device, such as serial-number, hardware-model, vendor, OS version, etc. If you are using the python interactive interpreter, you can see all available information, for example:

```
>>> dev = aos.Devices['08002737C2C1']
>>> dev
{
   "name": "08002737C2C1",
   "value": {
      "device_key": "08002737C2C1",
      "facts": {
         "hw_version": "",
         "mgmt_ifname": "eth0",
         "mgmt_macaddr": "08:00:27:37:c2:c1",
         "os_version_info": {
            "major": "3",
            "build": "1",
            "minor": "1"
         },
         "aos_hcl_model": "Cumulus_VX",
         "serial_number": "08002737C2C1",
         "os_arch": "x86_64",
         "vendor": "Cumulus",
         "os_version": "3.1.1",
         "os_family": "Cumulus",
         "hw_model": "VX",
         "aos_server": "192.168.59.250",
         "aos_version": "AOS_1.1.0_OB.11",
         "mgmt_ipaddr": "192.168.60.16"
      },
      "status": {
         "hostname": "leaf-3",
         "blueprint_id": "30cd9032-35f2-4532-8543-dc24fc8ec7cd",
         "blueprint_active": true,
         "error_message": "",
         "device_start_time": "2017-02-09T21:52:58.606303Z",
         "domain_name": "",
         "pool_id": "default_pool",
         "fqdn": "leaf-3",
         "comm_state": "on",
         "agent_start_time": "2017-02-09T21:53:14.000000Z",
         "state": "IS-ACTIVE",
         "is_acknowledged": true
      },
      "id": "MDgwMDI3MzdDMkMx",
      "user_config": {
         "aos_hcl_model": "Cumulus_VX",
```

```
        "admin_state": "normal",
        "location": "leaf-3@demo-vpod-l3 is node leaf_3"
      }
   },
   "id": "MDgwMDI3MzdDMkMx"
}
```

You can also access the `value` property directly, for example:

```
>>> print "S/N: {}".format(dev.value['facts']['serial_number'])
S/N: 08002737C2C1
```

Device Item Properties

The aos-pyez library provides the following device item properties as a convenience:

- `state` - provides the status/state value
- `is_approved` - `True` if the device is approved for use, `False` otherwise
- `user_config` - provide acess to the user_config dictionary

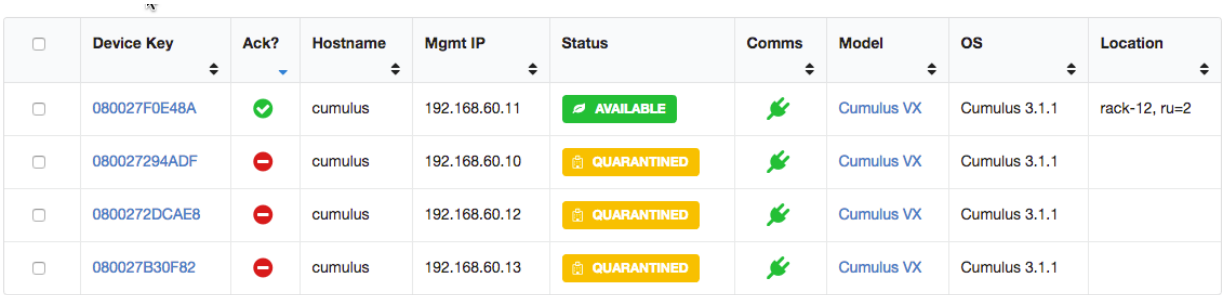
Approving Devices for Use

When a device initially registers with the AOS-Server (via the AOS device agent), the AOS-Server will place it into a Quarantined state. You are then required to approve it for use. Alternatively the you can pre-provision the AOS-Server with information so that a device will be recognized upon initial registration, skipping the approval step.

You can approve a given device via the aos-pyez library using the device instannce `approve()` method. This method takes an optional `location` parameter - this is an arbitrary string value you can use to identify where this device is located in the network, e.g. "rack-12, rack-unit-19". For more details on this API refer to *Devices*. Example:

```
>>> dev = aos.Devices['080027F0E48A']
>>> dev.is_approved
False
>>> dev.approve(location='rack-12, ru=2')
True
>>> dev.is_approved
True
```

After a device is approved, you can see the results on the AOS-Server UI, for example:



Device Key	Ack?	Hostname	Mgmt IP	Status	Comms	Model	OS	Location
080027F0E48A	✓	cumulus	192.168.60.11	AVAILABLE		Cumulus VX	Cumulus 3.1.1	rack-12, ru=2
080027294ADF	⊖	cumulus	192.168.60.10	QUARANTINED		Cumulus VX	Cumulus 3.1.1	
0800272DCAE8	⊖	cumulus	192.168.60.12	QUARANTINED		Cumulus VX	Cumulus 3.1.1	
080027B30F82	⊖	cumulus	192.168.60.13	QUARANTINED		Cumulus VX	Cumulus 3.1.1	

Collections

A Collection is a way to manage a group of similar items. The Collection base-class is used by many of the other modules within the aos-pyez library. In general, you can use a Collection to do the following:

- Get a list of known names in the collection
- Determine if an item exists in the collection
- Manage a specific item by the user defined item name (aka "label")
- Manage a specific item by the AOS unique id value (aka "uid")
- Iterate through each item in the collection

Collection Properties

The following properties are commonly used:

- `names` - provides a list of names known to the AOS-Server
- `LABEL` - the collection item property that identifies the User name for the item
- `UNIQUE_ID` - the collection item property that idetnifies the AOS-Server UID value

Accessing A Collection

You can access a collection of items via the Session instance. You can see a list of available collections from the as described in the *Client Session* guide page. For example, accessing the collection of IP Pools, you would access the `IpPools` session property:

```
>>> ip_pools = aos.IpPools
```

List of Known Items in a Collection

Each collection maintains a digest of information from the AOS-Server. This information is demand loaded when you access the collection. The digest contains the list of known User names ("labels") as well as some information about each of the items. The specific information that is collected and known is dependent on the collection. Refer to documentation on each of the collections for more details. To see a list of known collection labels, you can access the collection `names` property. For example, the AOS-server presently knows about the following IP Pools:

```
>>> aos.IpPools.names
[u'Switches-IpAddrs', u'Servers-IpAddrs']
```

Accessing a Collection Item

You can access a specific collection item in one of a few ways. The first way is by indexing the collection by the label name. For example, accessing the IP Pool named "Switches-IpAddrs":

```
>>> this_pool = aos.IpPools["Switches-IpAddrs"]
>>> this_pool.exists
True
>>> this_pool.id
u'65dfbc77-1c77-4a99-98a6-e36c5aa7e4d0'
```

If you attempt to access a collection item that does not exist using this method, you will still get an instance of a collection item, but this item does not yet exist in the AOS-Server. For example:

```
>>> new_pool = aos.IpPools['my_new_pool']
>>> new_pool.exists
False
>>> new_pool.id
>>>  # None
```

Generally the above approach is used when you want to create a new instance of a collection item. The topic of adding and removing collection items is covered in a following section.

Finding an Item in a Collection

Another way to access, or attempt to access, a collection item is using the collections `find()` method. This will either return an item that exists, or `None`. The `find()` method allows you on of two approaches; either find an item by its name or unqiue-id. For example:

```
>>> this_pool = aos.IpPools.find(label=u'Switches-IpAddrs')
>>> this_pool.id
>>> this_pool.id
u'65dfbc77-1c77-4a99-98a6-e36c5aa7e4d0'
```

Let's say that all you have is the UID value, perhaps from another API call, and you need to find the IP Pool for that UID. You can find it by using the `uid` argument, for example:

```
>>> pool = aos.IpPools.find(uid="65dfbc77-1c77-4a99-98a6-e36c5aa7e4d0")
>>> pool.exists
True
>>> pool.name
u'Switches-IpAddrs'
```

If you attempt to find an item that does not exist, by either `label` or `uid`, the `find()` method will return None.

```
>>> pool = aos.IpPools.find(uid="does not exist")
>>> pool is None
True
```

Checking for Item in Collection

If you simply want to determine if an item exists in the collection, i.e. known to the AOS-Server, you can use the `in` operator. For example, let's say you want to see if the IP Pool called "MyPool" is known to the AOS-Server:

```
>>> "MyPool" in aos.IpPools
False
```

This means that the AOS-Server does not manage this item.

Warning: The collection will only report on items that are known to the AOS-Server. So if you are in the process of creating a new collection item, but have not yet saved it to the AOS-Server, then the collection will still report that the item is not in the collection.

Iterating through Collection Items

If you need to loop through each item in a collection, you can do this using any pythonic iteration mechanism because the Collection base-class implements the iteration protocol. So you can do things like this:

```
>>> for pool in aos.IpPools:
...     print pool.name, pool.id
...
Switches-IpAddrs 65dfbc77-1c77-4a99-98a6-e36c5aa7e4d0
Servers-IpAddrs 0310d821-d075-4075-bdda-55cc6df57258
```

Adding and Removing Collection Items

The Collection base-class supports the `__iadd__()` and `__isub__()` operators. This is one way you can add and remove items. Other methods are described in the Collection-Item guide document.

Updating Collection Digest

If you need to update the aos-pyez collection data from the AOS-Server, for example, you're anticipating a change to the AOS-Server outside your program, then you can invoke the collection `digest()` method. This method will query the AOS-Server for what it knows, and rebuild the internal collection cache.

Pretty-Printing

Each collection implements the `__str__()` operator so you can pretty-print information about the collection. This is useful for interatice python sessions or general debugging. For example, here is the output for the IP Pools collection:

```
1  >>> aos.IpPools
2  {
3     "url": "resources/ip-pools",
4     "by_id": "id",
5     "item-names": [
6        "Switches-IpAddrs",
7        "Servers-IpAddrs"
8     ],
9     "by_label": "display_name"
10 }
```

Breaking down the above information:

- line 3: this is the URL in the AOS-Server API to access this collection
- line 4: the `id` is the actual property name within the collection item to provide the UID value
- lines 5-8: the list of known names managed by the AOS-Server
- line 9: the `display_name` is the actual property name within the collection item to provide the label value

Accessing the AOS-Server API Directly

The following properties are used if you need to access the AOS-Server API directly.

- `url` - This is the AOS-Server specific URL for this collection

- `api` - This is the Session instance so you can access the AOS-Server API

For example, here is the way you could directly perform a GET on the IP Pools collection:

```
>>> aos.IpPools.url
'http://aos-server:8888/api/resources/ip-pools'
```

```
>>> got = aos.IpPools.api.requests.get(aos.IpPools.url)
>>> got
<Response [200]>
```

Note: You do *not* need to provide the Requests header value to the `requests.get` call because the aos-pyez Session api instance has these values already stored within the session instance.

Collection Items

You can manage individual collection items in generally the same way. This guide page provides general usage information. Specific aos-pyez collections may have additional information that you can review as well. For more information about aos-pyez collections: *Collections*.

Item Properties

The following are commonly used collection item properties:

- `name` - This is the User provided name of the item, aka "label"
- `id` - This is the AOS-Server generated unique-ID value, aka "uid"
- `value` - This is a dict of data specific to the collection item that stores the raw data about this item.
- `collection` - This is the parent collection instance for this item.

Create an Item

You can create a new item in one of two ways. The first way is to access a collection using the new item name and then issuing a `write()` on the item. The `write()` will detect that this item does not currently exist in the AOS-Server and make the proper API call to create it. There is an explicit `create()` method that you could call in this particular use-case, but it is there for your programming convenience only.

For example, let's create a new IP Pool called "pod-1-switch-loopbacks". The first step is to index the IP Pools collection:

```
>>> new_pool = aos.IpPools['pod-1-switch-loopbacks']
>>> new_pool.exists
False
```

The next step is to provide the necessary item value data for this item. The structure / contents of the item data is going to be specific to each type of item. For specific item details, you will need to refer to the AOS-Server API Reference documentation available directly from the UI page.

You cannot write directly to the item `value` property, but you can provide the contents when you do the `write()` invocation:

```
# setup a dict of data required for the ip-pool item:
>>> pool_data = dict(subnets=[dict(network='192.168.10.0/24')])

# write the data, which will trigger a create
>>> new_pool.write(pool_data)
```

Upon success, the new pool now exists, and has been assigned a unique ID by AOS-Server. This information is updated within collection and item instance for your immediate use:

```
>>> new_pool.exists
True
>>> new_pool.id
u'45de5b41-1846-4057-afe8-9f5b93f8c5a6'
```

If for any reason the `write()` fails, an exception will be raised. So you should generally wrap a try/except around any operation you are doing with the aos-pyez library. For exception details, please refer to the reference section: *API Reference*.

Write an Item

If you need to create or update an existing collection item, you can do so using the `write()` method. If the item does not exist in the AOS-Server, then this method will perform the necessary POST command to create it. Otherwise, the `write()` method will issue a PUT command to update-overwrite.

Read an Item

Generally speaking, when you access a collection item, the item value is already present. If you need for any reason to retrieve the current value from the AOS-Server, you can invoke the `read()` method. This will refresh the instance value.

Delete an Item

You can delete an item in one of two ways - either calling the `delete()` method on the instance or using the python `del` operation on the item `value` property.

```
#
# delete item using the method
#
>>> new_pool.delete()
#
# equivalent to using the del operator
#
>>> del new_pool.value
```

Backup / Restore Local JSON File

You may find it useful to make a copy of a collection item and store it as a JSON file on your local filesystem. You can then later restore this value from your local filesystem. Each collection item provides a `jsonfile_save()` and `jsonfile_load()` for backup and restore. By default, the `jsonfile_save()` will store the JSON file in your local current working directory using the `name` property as the filename. You can override this default behavior with the various arguments to the `jsonfile_save()` method.

```
>>> # assume new_pool was created with name='pod-1-switch-loopbacks'

>>> new_pool.jsonfile_save()                                    # saves to 'pod-1-
↪switch-loopbacks.json' in $CWD
>>> new_pool.jsonfile_save(dirpath='/tmp')                      # /tmp/pod-1-switch-
↪loopbacks.json
>>> new_pool.jsonfile_save(dirpath='/tmp', filename='save-me')   # /tmp/save-me.json
```

The `jsonfile_load()` method always requires a specific filepath:

```
>>> new_pool.jsonfile_load('/tmp/pod-1-switch-loopbacks.json')
```

Note: The `jsonfile_load()` method only loads the contents of the file into the instance object. If you are using this method to create a new item in the AOS-Server, you will then need to issue the `write()` method

Accessing the AOS-Server API Directly

In some cases you might want to access the AOS-Server API directly. The following properties are available should you need to do so:

- `url` - This is the AOS-Server specific URL for this item
- `api` - This is the Session instance so you can access the AOS-Server API

For example, let's say you want to issue a DELETE command directly:

```
>>> aos.IpPools.names
[u'Switches-IpAddrs', u'Servers-IpAddrs', u'my_pool']
>>>
>>> pool = aos.IpPools['my_pool']
>>>
>>> pool.url
u'http://aos-server:8888/api/resources/ip-pools/a91d088f-ee0e-4bfc-803f-9078954d5826'
>>>
>>> pool.api.requests.delete(pool.url)
<Response [202]>
```

Resources

This version of the aos-pyez supports all AOS 1.1 managed Resources. Resources are collections of data items that you use with AOS Blueprints for the purpose assigning values for network services.

- IP Pools
- ASN Pools
- External Routers

IP Pools

When you need to assign IP addresses to an AOS Blueprint you can use IP Pools to instruct AOS to allocate and assign addresses from these pool rather than explicitly assigning specific individual values.

You can use AOS IP Pools in conjunction with your existing IP Address Management (IPAM) system. An IPAM system allows you to allocate blocks of IP address ranges. If you use an IPAM system you know that you are still responsible for manually assigning specific IP addresses from those ranges when you create your network services. When you use AOS, you can create an AOS IP Pool with the IPAM range information, and then assign the IP Pool to an AOS Blueprint. By following this approach you let AOS do the specific IP allocation task.

You can access the IP Pools resources via the aos-pyez library by using the `IpPools` property of the Session:

```
>>> aos.IpPools
```

The IpPools is managed as a Collection/Items, as described in the *Collections* guide documents. In addition to the Items properties documented there, the an IpPool item also supports:

- `in_use` - `True` if the pool is used by an AOS Blueprint, `False` otherwise.

ASN Pools

When you need to assign Autonomous System Numbers (ASNs) to an AOS Blueprint you can use ASN Pools to instruct AOS to allocate and assign values from these pool rather than explicitly assigning specific individual values.

You can use AOS ASN Pools in conjunction with your existing IP Address Managementq (IPAM) system, presuming it also supports ASN block management. Even though an IPAM system allows you to allocate ASN blocks, you are still responsible for manually assigning specific values to devices when you create your network services. When you use AOS, you can create an AOS ASN Pool with the block range information from your IPAM. You then assign that ASN Pool into your Blueprint. By following this approach you let AOS do the specific ASN value assignment task.

You can access the ASN Pools resources via the aos-pyez library by using the `AsnPools` property of the Session:

```
>>> aos.AsnPools
```

The AsnPools is managed as a Collection/Items, as described in the *Collections* guide documents. In addition to the Items properties documented there, the an IpPool item also supports:

- `in_use` - `True` if the pool is used by an AOS Blueprint, `False` otherwise.

External Routers

AOS Blueprints need information about external routers. For example, when using AOS to manage a L3 Clos , the Blueprint needs external router information to configure the attached switches and have the necessary information to gather the appropriate telemetry associated with the BGP session(s).

You can access the External Routers resource via the aos-pyez library using the `ExternalRouters` property of the Session:

```
>>> aos.ExternalRouters
```

The ExternalRouters is managed as a Collection/Items, as described in the *Collections* guide documents.

Design Elements

Design elements are used when creating network service designs, called *Templates*. These templates are then used as a basis for specific instances of managed network services called *Blueprints*. For details on managing Blueprints, refer to the guide document *Blueprints*. This guide page is meant to provide information on using the AOS design elements

via the aos-pyez library. For complete documentation on these topics please refer to the AOS product documentation available via the support portal.

This version of the aos-pyez supports all AOS 1.1 managed design elements:

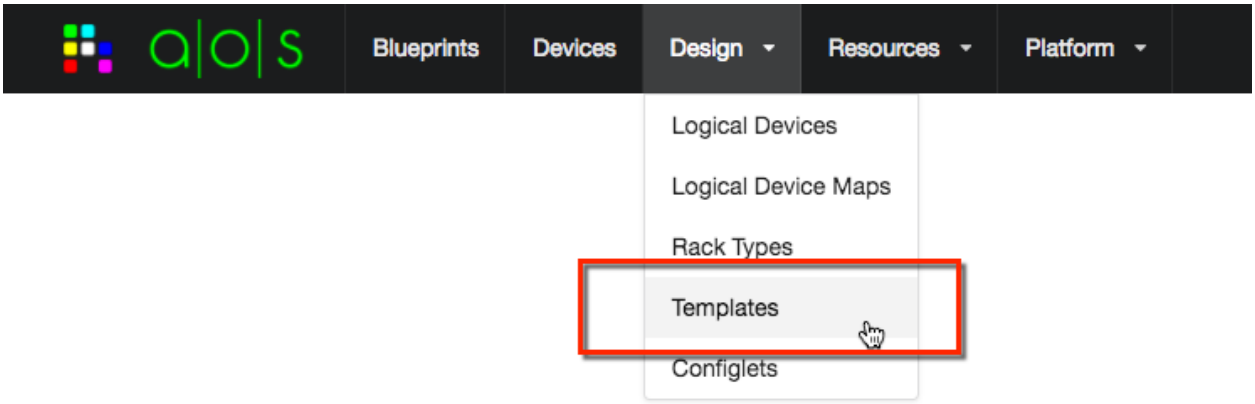
- Design Templates
- Logical Devices
- Logical Device Maps
- Rack Types

For this version of aos-pyez you are able to manage design elements as a collection, as described in the *Collections* guide pages. As a practical matter, you would use the AOS-Server UI to create these elements. You can then use the aos-pyez library to use the Template designs as part of building a Blueprint. Alternatively you could use the aos-pyez library to load the item contents from saved JSON files and create them in the AOS-Server.

Design Templates

When you create a design template you are instructing AOS on how to compose a network service in a vendor agnostic manner - meaning the design itself does not include any network vendor hardware or network operating system specific requirements. You assign those details later when you build out a Blueprint based on the design template.

For example, a design Template may describe a two stage L3 Clos data center fabric that uses 32x40 GE spine-switches . The design itself is vendor agnostic as it constrains the design to specific device property (32x40GE) but not to a specific network equipment vendor or network operating system (NOS). When you create a Blueprint from a Template, you can then assign into that Blueprint any specific vendor that provides for such a switch. You create these design / build relationships using the other design elements such as Logical Devices, Logical Device Maps, and Rack Types.



You can access the design Templates via the aos-pyez library by using the `DesignTemplates` property of the Session:

```
>>> aos.DesignTemplates
```

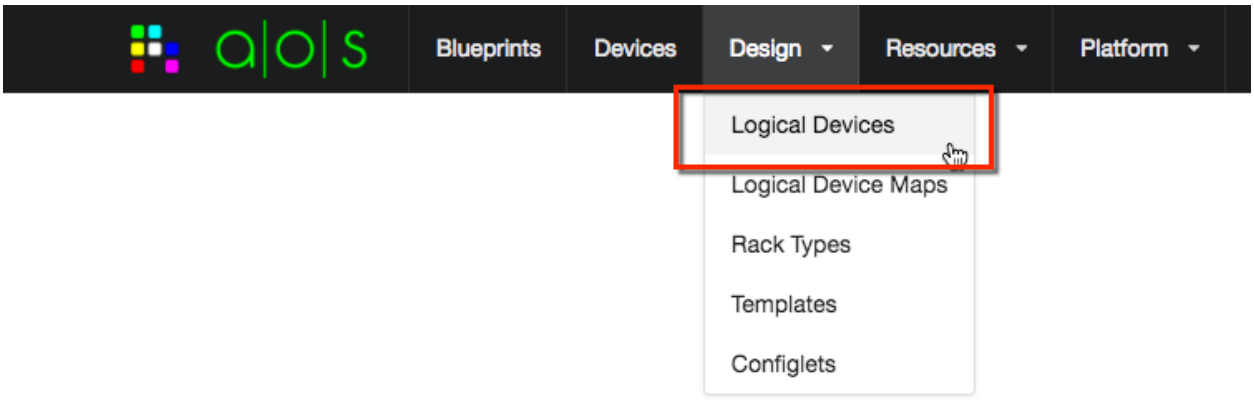
Logical Devices

You use a Logical Device as a design element for a *Design Templates* or *Rack Types*. You use Logical Devices to define the device properties that will then be used as part of a design. A logical device design construct is vendor agnostic, meaning that the properties defined are not speciifc to any one network equipment vendor or NOS.

Broadly speaking, these properties include the following:

- Number of ports - ports can also be arranged into groups
- Speed of ports - 10GE, 40GE, 100GE
- Role of ports - connects leaf to spine, connect to external-router, to-attached server, etc.

A typical example of a logical device is a a leaf switch with 48x10GE + 6x40GE ports. The 48x10GE could be designated to connect to either servers or external-routers (for the case of 10GE router connections). The 6x40GE ports would be designated as connecting leaf to spine, i.e. L3 Clos fabric ports.



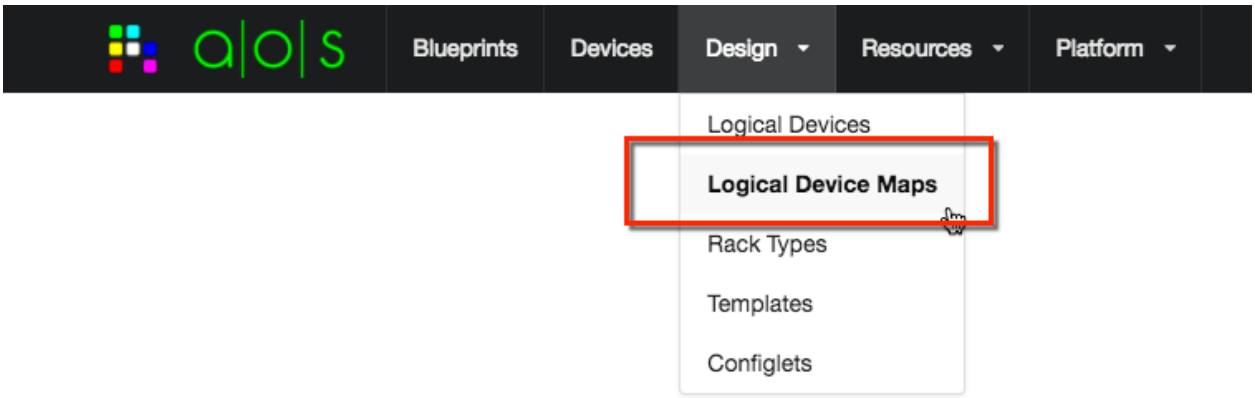
You can access the Logical Devices via the aos-pyez library by using the `LogicalDevices` property of the Session:

```
>>> aos.LogicalDevices
```

Logical Device Maps

You use a Logical Device Map to create the relationship between a Logical Device (vendor agnostic) and a specific network vendor equipment running a specific NOS. You then use logical device maps when building a Blueprint to identify the specific network equipment to use.

For example, you might have a design that requires a 32x40GE spine, as defined by a logical device. You would then need to create a logical device map for the specific equipment you plan to use. If you want to use a Dell S-6000 switch running Cumulus Linux, you would create a specific logical device map for that purpose. Likewise, if you wanted to use a Cisco 9332 running NX-OS you would create another logical device for that purpose. Once you have logical device maps created, you can then use these to associate specific vendor equipment into a Blueprint.



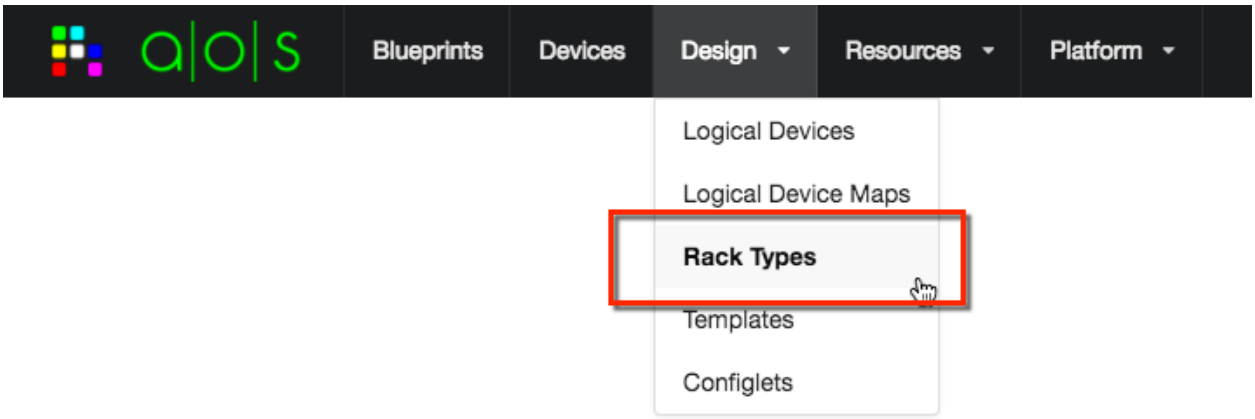
You can access the Logical Device Maps via the aos-pyez library by using the `LogicalDeviceMaps` property of the Session:

```
>>> aos.LogicalDeviceMaps
```

Rack Types

You use Rack Types as an element of a design Template. You can compose the structure of rack that includes the following properties:

- Number of leaf-switches per rack
- Logical Device type of leaf-switches
- Number of links between leaf-switches within the rack (for redundancy)
- Number of links between leaf-switches and spine-switches
- Number of servers within the rack
- Logical Device type of servers (modeling the number NICs)
- The manner in which the server NICs are connected between rack leaf-switches



You can access the Logical Device Maps via the aos-pyez library by using the `RackTypes` property of the Session:

```
>>> aos.RackTypes
```

Blueprints

A Blueprint is a specific instance of an engineering Design Template that you build, deploy, and operate within your network. A typical use-case is the following:

1. create a blueprint from a given design template
2. build out the blueprint with specific parameter values
3. assign specific devices to the blueprint
4. deploy the blueprint onto the devices
5. obtain operational values about the blueprint

This version of the aos-pyez library focuses the "build" aspects; meaning you can use the library to create a Blueprint from a design Template and provision the Blueprint parameters. Future versions of the aos-pyez library will expose the complete set of functionality. You can access all of the Blueprint functionality using the method described in the section *Session.api.requests*.

You can access the Blueprints via the aos-pyez library by using the `Blueprints` property of the Session:

```
>>> aos.Blueprints
```

Supported Features

Create a Blueprint from a Design Template

Let's create a Blueprint from a design Template called "My-Demo-Design". We can verify that you've created this template using the `DesignTemplates` Session property:

```
>>> aos.DesignTemplates
{
   "url": "design/templates",
   "by_id": "id",
   "item-names": [
      "My-Demo-Design",
      "vPOD-L3"
   ],
   "by_label": "display_name"
}
```

To create a Blueprint, you must have the design Template **id** value, along with the name of the design reference architecture. For this example, the reference architecture is called *two_stage_l3clos*:

```
>>> template = aos.DesignTemplates['My-Demo-Design']
>>> template.id
u'ee0164ed-d9cf-46c8-a5e5-8b16d70d0a1c'
```

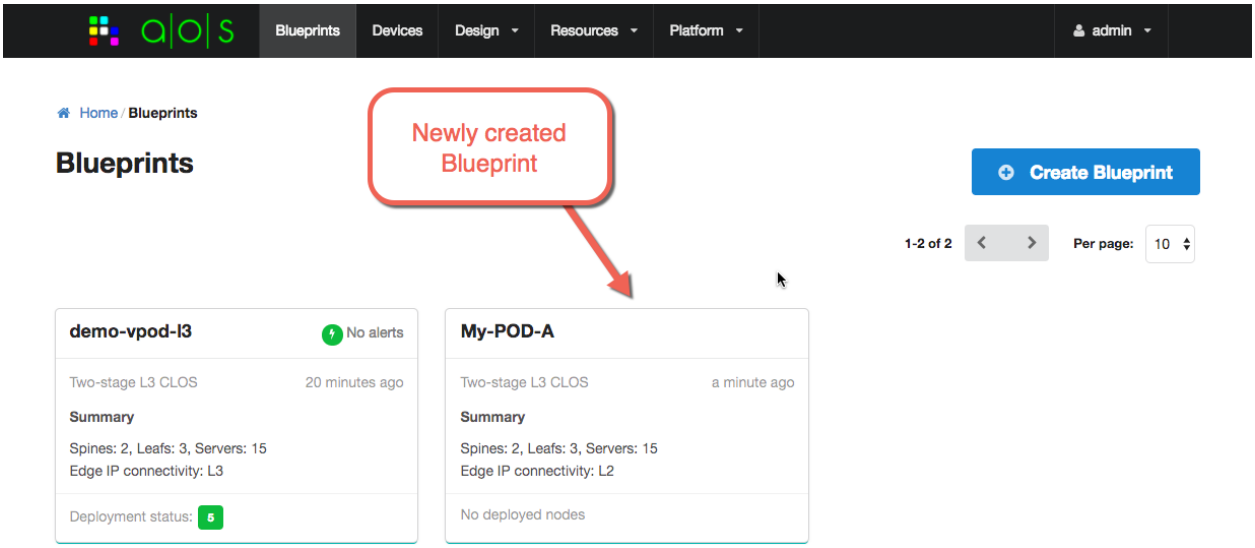
Let's create a Blueprint called "My-Pod-A" from this design. The first step is to create a blueprint item using the collection:

```
>>> blueprint = aos.Blueprints['My-POD-A']
>>> blueprint.exists
False
```

We can see that AOS-Server does not yet know about this Bluerpint because the `exists` property is `False`. The next step is to perform the create action:

```
>>> blueprint.create(template.id, reference_arch='two_stage_l3clos')
True
>>> blueprint.exists
True
>>> blueprint.id
u'a58e8c3f-84c5-472c-aaf8-a2292f4aa2c6'
```

At this point the blueprint exists in the AOS-Server, and you can verify that via the UI, for example:



Get / Set Blueprint Parameters

Once you have created a Blueprint, you will need to assign values that are specific to this service. These values include, for example, the IP addresses, VLANs, ASNs, specific device-models to use, and ultimately specific devices. You will need to provide all blueprint values before you can actually deploy the blueprint to the devices. This section describes the generalized process to get and set blueprint parameters. If you are specifically looking to manage the blueprint links, see *Cabling*.

You can assign blueprint values using the designated *slots* defined by the design Template. You can get a list of the known slots using the `params.names` list. For example:

```
>>> print json.dumps(blueprint.params.names, indent=2)
[
  "external_router_interfaces",
  "leaf_loopback_ips",
  "spine_leaf_link_ips",
  "to_external_router_link_ips",
  "leaf_asns",
  "node_leaf_1_interfaces",
  "hcls",
  "devices",
  "node_spine_1_interfaces",
  "spine_loopback_ips",
  "deploy",
  "spine_asns",
  "dhcp_server_ip",
  "logical_device_maps",
  "resource_pools",
  "node_leaf_3_interfaces",
  "node_leaf_2_interfaces",
  "hostnames",
  "external_links",
  "node_spine_2_interfaces",
  "port_maps"
]
```

You can inspect the purpose and current value of a blueprint parameter by accessing it as a collection item. For example, let's look at a commonly used slot called *resource_pools*. This parameter is where you would assign the specific IP-Pools and ASN-Pools for this blueprint.

```
>>> param = blueprint.params['resource_pools']
>>> param
{
   "Blueprint Name": "My-POD-A",
   "Blueprint ID": "a58e8c3f-84c5-472c-aaf8-a2292f4aa2c6",
   "Parameter Name": "resource_pools",
   "Parameter Value": {},
   "Parameter Info": {
      "slot_type": "POOL_LIST",
      "name": "resource_pools",
      "ids": [
         "leaf_loopback_ips",
         "spine_leaf_link_ips",
         "to_external_router_link_ips",
         "leaf_asns",
         "spine_loopback_ips",
         "spine_asns",
         "virtual_network_svi_subnets"
      ]
   }
}
```

The **resource_pool** is one of the more complicated, but most often used, values. So it's worthwhile to explore this one in detail here. You can see from the "Parameter Info" description that the *resource_pools* slot has a number of sub-parameters, *leaf_asns* for example. This means that if you want to assign the ASN-Pool for the leaf switches, you would store that value there. Looking at the "Parameter Value" area, you can see that no values are actually assigned to the *resource_pools* slot. So let's assign the ASN pool called "Private-ASN-pool":

```
>>> aos.AsnPools['Private-ASN-pool'].id
u'b4fdb577-531b-40ba-96a8-9a015794b30c'
```

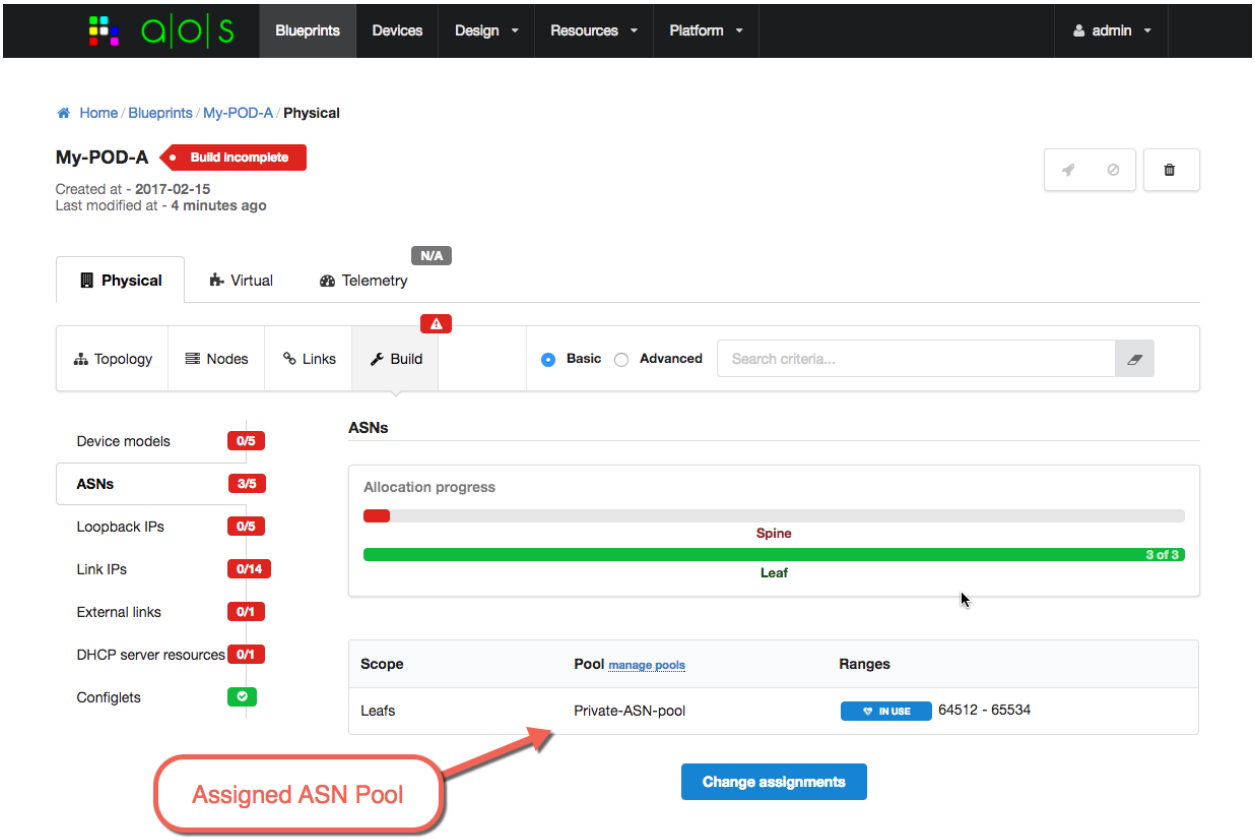
From the "slot_type" information we can see the value must actually be a **list**. So in order to update one specific slot in the **resource_pools** you would need to do something like this:

```
>>> param.update({'leaf_asns': [aos.AsnPools['Private-ASN-pool'].id]})
```

You can see `update()` takes a value to merge into the slot, and that value is a dictionary keyed by the slot ids . In this case we are only updating a single key, **leaf_asns**. The value for this key must be a **list** of ASN-Pool IDs. In this case, we are only assigning a single pool, so a list of one element. Once this action is completed, you can verify this by examining the param `value` property:

```
>>> param.value
{u'leaf_asns': [u'b4fdb577-531b-40ba-96a8-9a015794b30c']}
```

You can also see this update on the AOS-Server UI, for example:



Let's look at one more example, this one much less complex. This blueprint has a DHCP relay service component, and one of the blueprint parameters is to provide the DHCP server IP address. The slot name here is:

```
>>> blueprint.params['dhcp_server_ip']
{
   "Blueprint Name": "My-POD-A",
   "Blueprint ID": "a58e8c3f-84c5-472c-aaf8-a2292f4aa2c6",
   "Parameter Name": "dhcp_server_ip",
   "Parameter Value": {},
   "Parameter Info": {
      "slot_type": "IP",
      "name": "dhcp_server_ip",
      "ids": [
         "value"
      ]
   }
}
```

So in order to assign this value, we would need to perform a parameter update, the key is **value** and we need to provide the IP address. We could do this update in a simple line, for example:

```
>>> blueprint.params['dhcp_server_ip'].update({'value': "192.168.59.254"})
```

And reading this back, we can see the update is completed as reflected in the "Parameter Value":

```
>>> blueprint.params['dhcp_server_ip']
{
   "Blueprint Name": "My-POD-A",
   "Blueprint ID": "a58e8c3f-84c5-472c-aaf8-a2292f4aa2c6",
```

```
   "Parameter Name": "dhcp_server_ip",
   "Parameter Value": {
      "value": "192.168.59.254"
   },
   "Parameter Info": {
      "slot_type": "IP",
      "name": "dhcp_server_ip",
      "ids": [
         "value"
      ]
   }
}
```

or more simply by examining just the `value` property:

```
>>> blueprint.params['dhcp_server_ip'].value
{u'value': u'192.168.59.254'}
```

For more details on using the aos-pyez API with Blueprints, see *Blueprints*.

Get / Set Blueprint Cabling

New in version 0.7.0.

You may want to manage the Blueprint cabling. Blueprint instances now support a `cabling` property that provides an easy way to retrieve and and reconfigure the cabling parameters. This `cabling` property builds upon the general methodology presented in the *Blueprint Parameters*.

Current documentation is provided in the *Blueprint Cabling API* section.

Retrieve Blueprint Build Status

You can check to see if the blueprint has any missing build values by examinging the `build_errors` property. This property is a list of current missing build issues, or `None` if there are no build issues. Here is a short listing of the system related issues for a new blueprint, for example:

```
>>> print json.dumps(blueprint.build_errors['system']['nodes'], indent=2)
{
  "leaf_3": {
    "hcl_id": "Value should be set",
    "port_map": "Value should be set",
    "loopback_ip": "Value should be set"
  },
  "leaf_2": {
    "hcl_id": "Value should be set",
    "port_map": "Value should be set",
    "loopback_ip": "Value should be set"
  },
  "leaf_1": {
    "hcl_id": "Value should be set",
    "loopback_ip": "Value should be set",
    "port_map": "Value should be set"
  },
  "spine_2": {
    "hcl_id": "Value should be set",
    "loopback_ip": "Value should be set",
```

```
    "port_map": "Value should be set",
    "asn": "Value should be set"
  },
  "spine_1": {
    "hcl_id": "Value should be set",
    "loopback_ip": "Value should be set",
    "port_map": "Value should be set",
    "asn": "Value should be set"
  }
}
```

Retrieve Blueprint Rendered Contents

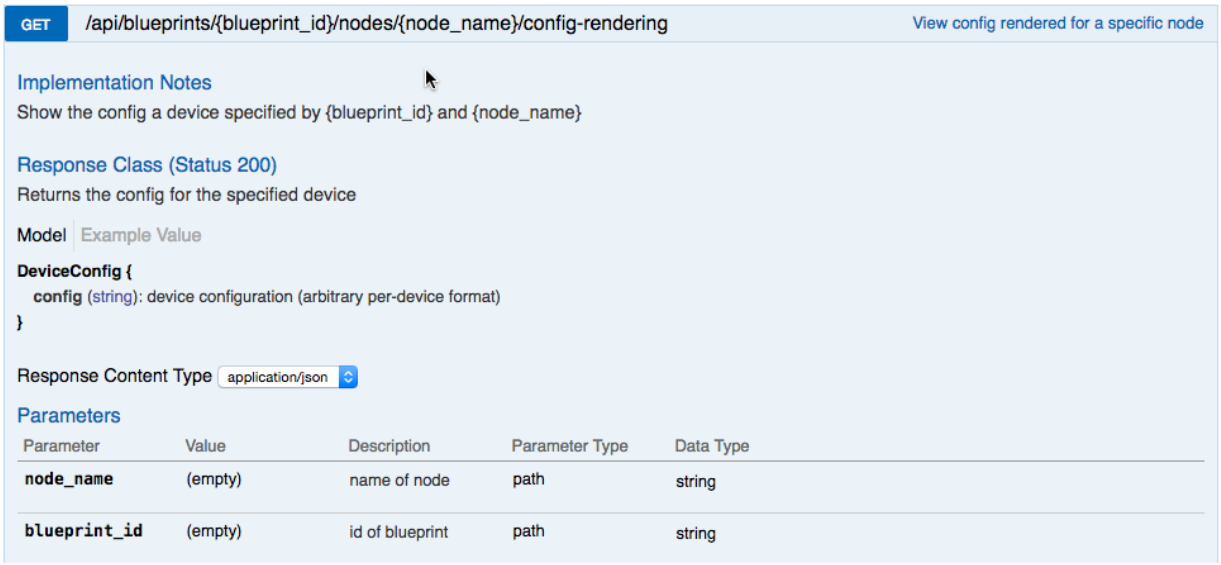
You can retrieve the contents of the Blueprint build composition at any time using the `contents` property. This property will give you a very large dictionary of data organized in a way that is specific to the design Template. The following is a short snippet of content values for the "links" area:

```
# each time you access the contents property the library will do a new GET.
# so make a variable so we only do a single GET for this example.

>>> contents = blueprint.contents

# examine a list of all the items in the contents

>>> print json.dumps(contents.keys(), indent=2)
[
  "display_name",
  "reference_architecture",
  "service",
  "created_at",
  "custom_extension",
  "system",
  "last_modified_at",
  "intent",
  "tenant_connectivity",
  "external_endpoints",
  "id",
  "constraints"
]

# access the system links contents.  There are 23 links in this Blueprint

>>> links = contents['system']['links']
>>> len(links)
23

# now show the contents of one of the links

>>> print json.dumps(links[0], indent=2)
{
  "role": "leaf_l3_server",
  "endpoints": [
    {
      "interface": "eth0",
      "display_name": "server_4_leaf_2",
      "type": "l3_server",
```

```
      "id": "server_4_leaf_2",
      "ip": "172.21.0.17/31"
    },
    {
      "interface": "swp4",
      "display_name": "leaf_2",
      "type": "leaf",
      "id": "leaf_2",
      "ip": "172.21.0.16/31"
    }
  ],
  "display_name": "leaf_2<->server_4_leaf_2"
}
```

Retrieve Device Rendered Configurations

Once you've completed the build out of the Blueprint parameters, you can retrieve the actual equipment vendor specific configuration that AOS will deploy onto the device. The structure and format of the device configuration will be specific to the equipment+NOS. The following example is part of the configuration for an Arista EOS device. In this example, the "spine_1" device happens to be an Arista. The aos-pyez library does not a convenient method to retrieve the configuration, so we need to use the *Session.api.requests* technique. The AOS-Server API reference page for the required request is:



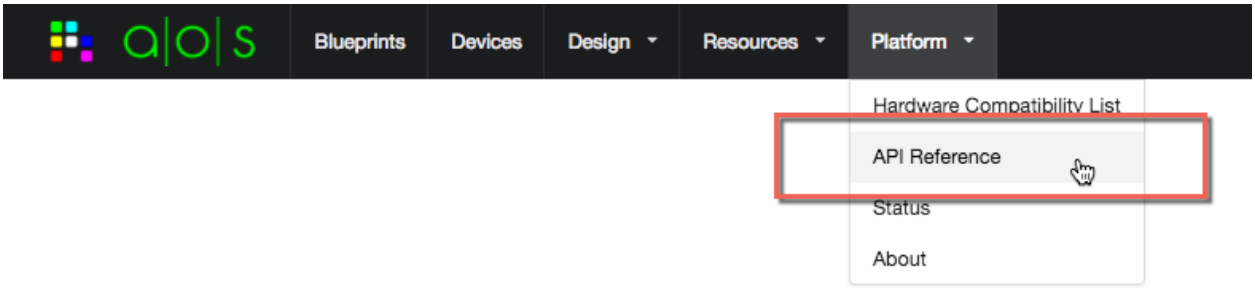
And the code example to invoke this API for the "spine_1" node:

```
>>> got = blueprint.api.requests.get("%s/nodes/spine_1/config-rendering" % blueprint.
↪url)
>>> body = got.json()
>>> body.keys()
[u'config']
>>>
>>> print body['config']
service interface inactive expose
!
hostname spine-1
interface Ethernet1
```

```
  description facing_router-A:
  no switchport
  ip address 1.1.1.0/31
  no shutdown
  exit
!
interface Ethernet2
  description facing_leaf_1:swp6
  no switchport
  ip address 172.20.0.0/31
  no shutdown
  exit
!
# ... clipped ...
```

More Features Soon!

Much of the Blueprint functionality from a *deploy* and *operate* phase is not currently exposed via the aos-pyez. Stay tuned for more enhancements in the coming releases. If you have any particular interests, please open a github issue. In the meantime, you can use the *Session.api.requests* technique illustrated above. You can find all of the AOS-Server API information directly from the UI:



CHAPTER 2

API Reference

Session

class apstra.aosom.session.**Session**(*target=None*, ***kwargs*)

The Session class is used to create a client connection with the AOS-server. The general process to create a connection is as follows:

```
from apstra.aosom.session import Session

aos = Session('aos-session')                    # hostname or ip-addr of AOS-server
aos.login()                                     # username/password uses defaults
```

This module will use your environment variables to provide the default login values, if they are set. Refer to *ENV* for specific values.

This module will use value defaults as defined in *DEFAULTS*.

Once you have an active session with the AOS-server you use the modules defined in the ModuleCatalog.

The following are the available public attributes of a Session instance:

- api - an instance of the Session.Api that provides HTTP access capabilities.
- server - the provided AOS-server hostname/ip-addr value.
- user - the provided AOS login user-name

The following are the available user-shell environment variables that are used by the Session instance:

- AOS_SERVER - the AOS-server hostname/ip-addr
- AOS_USER - the login user-name, defaults to DEFAULTS["USER"].
- AOS_PASSWD - the login user-password, defaults to DEFAULTS["PASSWD"].
- AOS_SESSION_TOKEN - a pre-existing API session-token to avoid user login/authentication.

ENV = {'USER': 'AOS_USER', 'PASSWD': 'AOS_PASSWD', 'TOKEN': 'AOS_SESSION_TOKEN', 'PORT': 'AOS_SE

DEFAULTS = {'PASSWD': 'admin', 'USER': 'admin', 'PORT': 443}

__init__(*target=None*, ***kwargs*)

Create a Session instance that will connect to an AOS-server, `server`. Additional keyword arguments can be provided that override the default values, as defined in *DEFAULTS*, or the values that are taken from the callers shell environment, as defined in *ENV*. Once a Session instance has been created, the caller can complete the login process by invoking *login()*.

Parameters

- **target** (*str*) – URL to the AOS-Server. The target value must be in the form of <scheme>://<aos-server>[:port]. For example:

  https://aos-server http://aos-server http://aos-server:8888

- **user** (*str*) – User login name
- **passwd** (*str*) – User login password

login()

Login to the AOS-server, obtaining a session token for use with later calls to the API.

Raises

- `LoginAuthError` – The provided user credentials are not valid. Check the user/password or session token values provided.
- `LoginServerUnreachableError` – The API is not able to connect to the AOS-server via the API. This could be due to any number of networking related issues. For example, the `port` is blocked by a firewall, or the `server` value is IP unreachable.
- `LoginNoServerError` – The instance does not have `server` configured.

session

When used as a `setter` attempts to resume an existing session with the AOS-server using the provided session data. If there is an error, an exception is raised.

Returns The session data that can be used for a future resume.

Return type dict

Raises See the *login()* for details.

token

Returns Authentication token from existing session.

Return type str

Raises `NoLoginError` – When no token is present.

url

Returns

- *Return the current AOS-server API URL. If this value is*
- *not set, then an exception is raised. The raise here is important*
- *because other code depends on this behavior.*

Raises NoLoginError: URL does not exist

Devices

Collection

`Collection` is the base-class for all collections managed by the AOS-PyEZ framework. Each of the collections managed within the AOS-PyEZ framework will subclass, and may provide additional functionality or override items in this base-class.

CollectionItem

`CollectionItem` is the base-class for all items that are stored within a collection. Each of the collections managed within the AOS-PyEZ framework will subclass, and may provide additional functionality or override items in this base-class.

Blueprints

Blueprint Collection

Blueprint Item

Blueprint Parameter Collection

Blueprint Parameter Collection Item

Blueprint Cabling

Node Cablers

The `NodeCabler` class is a helper to `Cabling`, refer to `Cabling.Nodes`.

Link Cablers

The `LinkCabler` class is a helper to `Cabling`, refer to `Cabling.Links`.

CHAPTER 3

Other information

Changelog

0.6.0 (14-Feb-2017)

- migrated to using requests library Session instance with apstra.aosom.Session.api
- unit-test coverage at 100%
- docs updated
- depreciated AOS 1.0 specific code
- bug fixes with some breaking-changes; refer to tagged release notes for details

License

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
```

```
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other modifications
      represent, as a whole, an original work of authorship. For the purposes
      of this License, Derivative Works shall not include works that remain
      separable from, or merely link (or bind by name) to the interfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or additions
      to that Work or Derivative Works thereof, that is intentionally
      submitted to Licensor for inclusion in the Work by the copyright owner
      or by an individual or Legal Entity authorized to submit on behalf of
      the copyright owner. For the purposes of this definition, "submitted"
      means any form of electronic, verbal, or written communication sent
      to the Licensor or its representatives, including but not limited to
      communication on electronic mailing lists, source code control systems,
      and issue tracking systems that are managed by, or on behalf of, the
      Licensor for the purpose of discussing and improving the Work, but
      excluding communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a Contribution."

      "Contributor" shall mean Licensor and any individual or Legal Entity
      on behalf of whom a Contribution has been received by Licensor and
      subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      copyright license to reproduce, prepare Derivative Works of,
      publicly display, publicly perform, sublicense, and distribute the
      Work and such Derivative Works in Source or Object form.

   3. Grant of Patent License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
```

```
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
```

```
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.
```

CHAPTER 4

Indices and Tables

- genindex
- modindex
- search

Index

Symbols

D

E

L

S

T

U