pylxd Documentation

Canonical Ltd

Jun 03, 2020

Contents

Contents:

CHAPTER 1

Installation

If you're running on Ubuntu Xenial or greater:

```
sudo apt-get install python-pylxd lxd
```

Otherwise you can track LXD development on other Ubuntu releases:

```
sudo add-apt-repository ppa:ubuntu-lxc/lxd-git-master && sudo apt-get update
sudo apt-get install lxd
```

Or install pylxd using pip:

```
pip install pylxd
```

CHAPTER 2

Getting started

2.1 Client

Once you have *installed*, you're ready to instantiate an API client to start interacting with the LXD daemon on localhost:

```
>>> from pylxd import Client
>>> client = Client()
```

If your LXD instance is listening on HTTPS, you can pass a two part tuple of (cert, key) as the *cert* argument.

```
>>> from pylxd import Client
>>> client = Client(
...     endpoint='http://10.0.0.1:8443',
...     cert=('/path/to/client.crt', '/path/to/client.key'))
```

Note: in the case where the certificate is self signed (LXD default), you may need to pass *verify=False*.

2.1.1 Querying LXD

LXD exposes a number of objects via its REST API that are used to orchestrate containers. Those objects are all accessed via manager attributes on the client itself. This includes *certificates*, *containers*, *images*, *networks*, *operations*, and *profiles*. Each manager has methods for querying the LXD instance. For example, to get all containers in a LXD instance

```
>>> client.containers.all()
[<container.Container at 0x7f95d8af72b0>,]
```

For specific manager methods, please see the documentation for each object.

2.1.2 pylxd Objects

Each LXD object has an analagous pylxd object. Returning to the previous *client.containers.all* example, a *Container* object is manipulated as such:

```
>>> container = client.containers.all()[0]
>>> container.name
'lxd-container'
```

Each pylxd object has a lifecycle which includes support for transactional changes. This lifecycle includes the following methods and attributes:

- *sync()* - Synchronize the object with the server. This method is called implicitly when accessing attributes that have not yet been populated, but may also be called explicitly. Why would attributes not yet be populated? When retrieving objects via *all*, LXD's API does not return a full representation.
- *dirty* - After setting attributes on the object, the object is considered "dirty".
- *rollback()* - Discard all local changes to the object, opting for a representation taken from the server.
- *save()* - Save the object, writing changes to the server.

Returning again to the *Container* example

```
>>> container.config
{ 'security.privileged': True }
>>> container.config.update({'security.nesting': True})
>>> container.dirty
True
>>> container.rollback()
>>> container.dirty
False
>>> container.config
{ 'security.privileged': True }
>>> container.config = {'security.privileged': False}
>>> container.save(wait=True)  # The object is now saved back to LXD
```

2.1.3 A note about asynchronous operations

Some changes to LXD will return immediately, but actually occur in the background after the http response returns. All operations that happen this way will also take an optional *wait* parameter that, when *True*, will not return until the operation is completed.

2.1.4 UserWarning: Attempted to set unknown attribute "x" on instance of "y"

The LXD server changes frequently, particularly if it is snap installed. In this case it is possible that the LXD server may send back objects with attributes that this version of pylxd is not aware of, and in that situation, the pylxd library issues the warning above.

The default behaviour is that *one* warning is issued for each unknown attribute on *each* object class that it unknown. Further warnings are then surpressed. The environment variable `PYLXD_WARNINGS` can be set to control the warnings further:

- if set to `none` then *all* warnings are surpressed all the time.
- if set to `always` then warnings are always issued for each instance returned from the server.

CHAPTER 3

Client Authentication

When using LXD over https, LXD uses an asymmetric keypair for authentication. The keypairs are added to the authentication database after entering the LXD instance's "trust password".

3.1 Generate a certificate

To generate a keypair, you should use the *openssl* command. As an example:

```
openssl req -newkey rsa:2048 -nodes -keyout lxd.key -out lxd.csr
openssl x509 -signkey lxd.key -in lxd.csr -req -days 365 -out lxd.crt
```

For more detail on the commands, or to customize the keys, please see the documentation for the *openssl* command.

3.2 Authenticate a new keypair

If a client is created using this keypair, it would originally be "untrusted", essentially meaning that the authentication has not yet occurred.

```
>>> from pylxd import Client
>>> client = Client(
...     endpoint='http://10.0.0.1:8443',
...     cert=('lxd.crt', 'lxd.key'))
>>> client.trusted
False
```

In order to authenticate the client, pass the lxd instance's trust password to *Client.authenticate*

```
>>> client.authenticate('a-secret-trust-password')
>>> client.trusted
>>> True
```

CHAPTER 4

Events

LXD provides an */events* endpoint that is upgraded to a streaming websocket for getting LXD events in real-time. The `Client`'s *events* method will return a websocket client that can interact with the web socket messages.

```
>>> ws_client = client.events()
>>> ws_client.connect()
>>> ws_client.run()
```

A default client class is provided, which will block indefinitely, and collect all json messages in a *messages* attribute. An optional *websocket_client* parameter can be provided when more functionality is needed. The *ws4py* library is used to establish the connection; please see the *ws4py* documentation for more information.

The stream of events can be filtered to include only specific types of events, as defined in the LXD /endpoint documentation.

To receive all events of type 'operation' or 'logging', generated by the LXD server:

```
>>> filter = set([EventType.Operation, EventType.Logging])
>>> ws_client = client.events(event_filter=filter)
```

To receive only events pertaining to the lifecycle of the containers:

```
>>> filter = set([EventType.Lifecycle])
>>> ws_client = client.events(event_filter=filter)
```

CHAPTER 5

Certificates

Certificates are used to manage authentications in LXD. Certificates are not editable. They may only be created or deleted. None of the certificate operations in LXD are asynchronous.

5.1 Manager methods

Certificates can be queried through the following client manager methods:

- *all()* - Retrieve all certificates.
- *get()* - Get a specifit certificate, by its fingerprint.
- *create()* - Create a new certificate. This method requires a first argument that is the LXD trust password, and the cert data, in binary format.

5.2 Certificate attributes

Certificates have the following attributes:

- *fingerprint* - The fingerprint of the certificate. Certificates are keyed off this attribute.
- *certificate* - The certificate itself, in PEM format.
- *type* - The certificate type (currently only "client")

CHAPTER 6

Containers

Container objects are the core of LXD. Containers can be created, updated, and deleted. Most of the methods for operating on the container itself are asynchronous, but many of the methods for getting information about the container are synchronous.

6.1 Manager methods

Containers can be queried through the following client manager methods:

- *exists(name)* - Returns *boolean* indicating if the container exists.
- *all()* - Retrieve all containers.
- *get()* - Get a specific container, by its name.
- *create(config, wait=False)* - Create a new container. This method requires the container config as the first parameter. The config itself is beyond the scope of this documentation. Please refer to the LXD documentation for more information. This method will also return immediately, unless *wait* is *True*.

6.2 Container attributes

For more information about the specifics of these attributes, please see the LXD documentation.

- *architecture* - The container architecture.
- *config* - The container config
- *created_at* - The time the container was created
- *devices* - The devices for the container
- *ephemeral* - Whether the container is ephemeral
- *expanded_config* - An expanded version of the config

- *expanded_devices* - An expanded version of devices
- *name* - (Read only) The name of the container. This attribute serves as the primary identifier of a container
- *description* - A description given to the container
- *profiles* - A list of profiles applied to the container
- *status* - (Read only) A string representing the status of the container
- *last_used_at* - (Read only) when the container was last used
- *status_code* - (Read only) A LXD status code of the container
- *stateful* - (Read only) Whether the container is stateful

6.3 Container methods

- *rename* - Rename a container. Because *name* is the key, it cannot be renamed by simply changing the name of the container as an attribute and calling *save*. The new name is the first argument and, as the method is asynchronous, you may pass *wait=True* as well.
- *save* - Update container's configuration
- *state* - Get the expanded state of the container.
- *start* - Start the container
- *stop* - Stop the container
- *restart* - Restart the container
- *freeze* - Suspend the container
- *unfreeze* - Resume the container
- *execute* - Execute a command on the container. The first argument is a list, in the form of *subprocess.Popen* with each item of the command as a separate item in the list. Returns a tuple of *(exit_code, stdout, stderr)*. This method will block while the command is executed.
- *raw_interactive_execute* - Execute a command on the container. It will return an url to an interactive websocket and the execution only starts after a client connected to the websocket.
- *migrate* - Migrate the container. The first argument is a client connection to the destination server. This call is asynchronous, so `wait=True` is optional. The container on the new client is returned. If `live=True` is passed to the function call, then the container is live migrated (see the LXD documentation for further details).
- *publish* - Publish the container as an image. Note the container must be stopped in order to use this method. If *wait=True* is passed, then the image is returned.
- *restore_snapshot* - Restore a snapshot by name.

6.4 Examples

If you'd only like to fetch a single container by its name…

```
>>> client.containers.get('my-container')
<container.Container at 0x7f95d8af72b0>
```

If you're looking to operate on all containers of a LXD instance, you can get a list of all LXD containers with *all*.

```
>>> client.containers.all()
[<container.Container at 0x7f95d8af72b0>,]
```

In order to create a new `Container`, a container config dictionary is needed, containing a name and the source. A create operation is asynchronous, so the operation will take some time. If you'd like to wait for the container to be created before the command returns, you'll pass *wait=True* as well.

```
>>> config = {'name': 'my-container', 'source': {'type': 'none'}}
>>> container = client.containers.create(config, wait=False)
>>> container
<container.Container at 0x7f95d8af72b0>
```

If you were to use an actual image source, you would be able to operate on the container, starting, stopping, snapshotting, and deleting the container. You can also modify container config (limits and etc).

```
>>> config = {'name': 'my-container', 'source': {'type': 'image', 'alias': 'ubuntu/
↪trusty'} 'config': {'limits.cpu': '2'}}
>>> container = client.containers.create(config, wait=True)
>>> container.start()
>>> container.freeze()
>>> container.delete()
```

Config line with a specific image source and a profile.

```
>>> config = {'name': 'my-container', 'source': {'type': 'image', "mode": "pull",
↪"server":
    "https://cloud-images.ubuntu.com/daily", "protocol": "simplestreams", 'alias':
↪'bionic/amd64'},
    'profiles': ['profilename'] }
```

To modify container's configuration method ' ' should be called after `Container` attributes changes.

```
>>> container = client.containers.get('my-container')
>>> container.ephemeral = False
>>> container.devices = { 'root': { 'path': '/', 'type': 'disk', 'size': '7GB'} }
>>> container.save()
```

To get state information such as a network address.

```
>>> addresses = container.state().network['eth0']['addresses']
>>> addresses[0]
{'family': 'inet', 'address': '10.251.77.182', 'netmask': '24', 'scope': 'global'}
```

To migrate a container between two servers, first you need to create a client certificate in order to connect to the remote server

> openssl req -newkey rsa:2048 -nodes -keyout lxd.key -out lxd.csr openssl x509 -signkey lxd.key -in lxd.csr -req -days 365 -out lxd.crt

Then you need to connect to both the destination server and the source server, the source server has to be reachable by the destination server otherwise the migration will fail due to a websocket error

```
from pylxd import Client

client_source=Client(endpoint='https://192.168.1.104:8443',cert=('lxd.crt','lxd.key'),
↪verify=False)
client_destination=Client(endpoint='https://192.168.1.106:8443',cert=('lxd.crt','lxd.
↪key'),verify=False)
```

(continues on next page)

(continued from previous page)

```
cont = client_source.containers.get('testm')
cont.migrate(client_destination,wait=True)
```

This will migrate the container from source server to destination server

To migrate a live container, user the `live=True` parameter:

```
cont.migrate(client__destination, live=True, wait=True)
```

If you want an interactive shell in the container, you can attach to it via a websocket.

```
>>> res = container.raw_interactive_execute(['/bin/bash'])
>>> res
{
    "name": "container-name",
    "ws": "/1.0/operations/adbaab82-afd2-450c-a67e-274726e875b1/websocket?
↪secret=ef3dbdc103ec5c90fc6359c8e087dcaf1bc3eb46c76117289f34a8f949e08d87",
    "control": "/1.0/operations/adbaab82-afd2-450c-a67e-274726e875b1/websocket?
↪secret=dbbc67833009339d45140671773ac55b513e78b219f9f39609247a2d10458084"
}
```

You can connect to this urls from e.g. https://xtermjs.org/ .

6.5 Container Snapshots

Each container carries its own manager for managing `Snapshot` functionality. It has *get*, *all*, and *create* functionality.

Snapshots are keyed by their name (and only their name, in pylxd; LXD keys them by <container-name>/<snapshot-name>, but the manager allows us to use our own namespacing).

A container object (returned by *get* or *all*) has the following methods:

- *rename* - rename a snapshot
- *publish* - create an image from a snapshot. However, this may fail if the image from the snapshot is bigger than the logical volume that is allocated by lxc. See https://github.com/lxc/lxd/issues/2201 for more details. The solution is to increase the *storage.lvm_volume_size* parameter in lxc.
- *restore* - restore the container to this snapshot.

```
>>> snapshot = container.snapshots.get('an-snapshot')
>>> snapshot.created_at
'1983-06-16T2:38:00'
>>> snapshot.rename('backup-snapshot', wait=True)
>>> snapshot.delete(wait=True)
```

To create a new snapshot, use *create* with a *name* argument. If you want to capture the contents of RAM in the snapshot, you can use *stateful=True*.

Note: Your LXD requires a relatively recent version of CRIU for this.

```
>>> snapshot = container.snapshots.create(
...     'my-backup', stateful=True, wait=True)
>>> snapshot.name
'my-backup'
```

6.6 Container files

Containers also have a *files* manager for getting and putting files on the container. The following methods are available on the *files* manager:

- *put* - push a file into the container.
- *get* - get a file from the container.
- *delete_available* - If the *file_delete* extension is available on the lxc host, then this method returns *True* and the *delete* method is available.
- *delete* - delete a file on the container.

Note: All file operations use *uid* and *gid* of 0 in the container. i.e. root.

```
>>> filedata = open('my-script').read()
>>> container.files.put('/tmp/my-script', filedata)
>>> newfiledata = container.files.get('/tmp/my-script2')
>>> open('my-script2', 'wb').write(newfiledata)
```

CHAPTER 7

Images

Image objects are the base for which containers are built. Many of the methods of images are asynchronous, as they required reading and writing large files.

7.1 Manager methods

Images can be queried through the following client manager methods:

- *all()* - Retrieve all images.
- *get()* - Get a specific image, by its fingerprint.
- *get_by_alias()* - Ger a specific image using its alias.

And create through the following methods, there's also a copy method on an image:

- *create(data, public=False, wait=True)* - Create a new image. The first argument is the binary data of the image itself. If the image is public, set *public* to *True*.
- *create_from_simplestreams(server, alias, public=False, auto_update=False, wait=False)* - Create an image from simplestreams.
- *create_from_url(url, public=False, auto_update=False, wait=False)* - Create an image from a url.

7.2 Image attributes

For more information about the specifics of these attributes, please see the LXD documentation.

- *aliases* - A list of aliases for this image
- *auto_update* - Whether the image should auto-update
- *architecture* - The target architecture for the image
- *cached* - Whether the image is cached

- *created_at* - The date and time the image was created
- *expires_at* - The date and time the image expires
- *filename* - The name of the image file
- *fingerprint* - The image fingerprint, a sha2 hash of the image data itself. This unique key identifies the image.
- *last_used_at* - The last time the image was used
- *properties* - The configuration of image itself
- *public* - Whether the image is public or not
- *size* - The size of the image
- *uploaded_at* - The date and time the image was uploaded
- *update_source* - A dict of update informations

7.3 Image methods

- *export* - Export the image. Returns a file object with the contents of the image. *Note: Prior to pylxd 2.1.1, this method returned a bytestring with data; as it was not unbuffered, the API was severely limited.*
- *add_alias* - Add an alias to the image.
- *delete_alias* - Remove an alias.
- *copy* - Copy the image to another LXD client.

7.4 Examples

`Image` operations follow the same protocol from the client's *images* manager (i.e. *get*, *all*, and *create*). Images are keyed on a sha-1 fingerprint of the image itself. To get an image...

```
>>> image = client.images.get(
...     'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855')
>>> image
<image.Image at 0x7f95d8af72b0>
```

Once you have an image, you can operate on it as before:

```
>>> image.public
False
>>> image.public = True
>>> image.save()
```

To create a new Image, you'll open an image file, and pass that to *create*. If the image is to be public, *public=True*. As this is an asynchonous operation, you may also want to *wait=True*.

```
>>> image_data = open('an_image.tar.gz', 'rb').read()
>>> image = client.images.create(image_data, public=True, wait=True)
>>> image.fingerprint
'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855'
```

CHAPTER 8

Networks

Network objects show the current networks available to LXD. Creation and / or modification of networks is possible only if 'network' LXD API extension is present.

8.1 Manager methods

Networks can be queried through the following client manager methods:

- *all()* - Retrieve all networks.
- *exists()* - See if a profile with a name exists. Returns *bool*.
- *get()* - Get a specific network, by its name.
- *create()* - Create a new network. The name of the network is required. *description*, *type* and *config* are optional and the scope of their contents is documented in the LXD documentation.

8.2 Network attributes

- `name` - The name of the network.
- `description` - The description of the network.
- `type` - The type of the network.
- `used_by` - A list of containers using this network.
- `config` - The configuration associated with the network.
- `managed` - *boolean*; whether LXD manages the network.

8.3 Profile methods

- *rename()* - Rename the network.
- *save()* - Save the network. This uses the PUT HTTP method and not the PATCH.
- delete() - Deletes the network.

8.4 Examples

Network operations follow the same manager-style as other classes. Networks are keyed on a unique name.

```
>>> network = client.networks.get('lxdbr0')

>>> network
Network(config={"ipv4.address": "10.74.126.1/24", "ipv4.nat": "true", "ipv6.address":
↪"none"}, description="", name="lxdbr0", type="bridge")

>>> print(network)
{
  "name": "lxdbr0",
  "description": "",
  "type": "bridge",
  "config": {
    "ipv4.address": "10.74.126.1/24",
    "ipv4.nat": "true",
    "ipv6.address": "none"
  },
  "managed": true,
  "used_by": []
}
```

The network can then be modified and saved.

```
>>> network.config['ipv4.address'] = '10.253.10.1/24'
>>> network.save()
```

To create a new network, use *create()* with a name, and optional arguments: *description* and *type* and *config*.

```
>>> network = client.networks.create(
...     'lxdbr1', description='My new network', type='bridge', config={})
```

```
>>> network = client.networks.create(
...     'lxdbr1', description='My new network', type='bridge', config={})
```

```
>>> network = client.networks.create(
...     'lxdbr1', description='My new network', type='bridge', config={})
```

CHAPTER 9

Profiles

Profile describe configuration options for containers in a re-usable way.

9.1 Manager methods

Profiles can be queried through the following client manager methods:

- *all()* - Retrieve all profiles
- *exists()* - See if a profile with a name exists. Returns *boolean*.
- *get()* - Get a specific profile, by its name.
- *create(name, config, devices)* - Create a new profile. The name of the profile is required. *config* and *devices* dictionaries are optional, and the scope of their contents is documented in the LXD documentation.

9.2 Profile attributes

- *config* - (dict) config options for containers
- *description* - (str) The description of the profile
- *devices* - (dict) device options for containers
- *name* - (str) name of the profile
- *used_by* - (list) containers using this profile

9.3 Profile methods

- *rename* - Rename the profile.
- *save* - save a profile. This uses the PUT HTTP method and not the PATCH.

- *delete* - deletes a profile.

9.4 Examples

`Profile` operations follow the same manager-style as Containers and Images. Profiles are keyed on a unique name.

```
>>> profile = client.profiles.get('my-profile')
>>> profile
<profile.Profile at 0x7f95d8af72b0>
```

The profile can then be modified and saved.

```
>>> profile.config.update({'security.nesting': 'true'})
>>> profile.devices.update({"eth0": {"parent": "lxdbr0", "nictype": "bridged", "type
↪": "nic", "name": "eth0"}})
>>> profile.save()
```

To create a new profile, use *create* with a name, and optional *config* and *devices* config dictionaries.

```
>>> profile = client.profiles.create(
...     'an-profile', config={'security.nesting': 'true'},
...     devices={'root': {'path': '/', 'size': '10GB', 'type': 'disk'}})
```

CHAPTER 10

Operations

Operation objects detail the status of an asynchronous operation that is taking place in the background. Some operations (e.g. image related actions) can take a long time and so the operation is performed in the background. They return an operation *id* that may be used to discover the state of the operation.

10.1 Manager methods

Operations can be queried through the following client manager methods:

- *get()* - Get a specific operation, by its id.
- *wait_for_operation()* - get an operation, but wait until it is complete before returning the operation object.

10.2 Operation object methods

- *wait()* - Wait for the operation to complete and return. Note that this can raise a *LXDAPIExceptiion* if the operations fails.

CHAPTER 11

Storage Pools

LXD supports creating and managing storage pools and storage volumes. General keys are top-level. Driver specific keys are namespaced by driver name. Volume keys apply to any volume created in the pool unless the value is overridden on a per-volume basis.

11.1 Storage Pool objects

`StoragePool` objects represent the json object that is returned from *GET /1.0/storage-pools/<name>* and then the associated methods that are then available at the same endpoint.

There are also `StorageResource` and `StorageVolume` objects that represent the storage resources endpoint for a pool at *GET /1.0/storage-pools/<pool>/resources* and a storage volume on a pool at *GET /1.0/storage-pools/<pool>/volumes/<type>/<name>*. Note that these should be accessed from the storage pool object. For example:

```
client = pylxd.Client()
storage_pool = client.storage_pools.get('poolname')
storage_volume = storage_pool.volumes.get('custom', 'volumename')
```

Note: For more details of the LXD documentation concerning storage pools please see LXD Storage Pools REST API Documentation and LXD Storage Pools Documentation. This provides information on the parameters and attributes in the following methods.

Note: Please see the pylxd API documentation for more information on storage pool methods and parameters. The following is a summary.

11.1.1 Storage Pool Manager methods

Storage-pools can be queried through the following client manager methods:

- *all()* - Return a list of storage pools.
- *get()* - Get a specific storage-pool, by its name.
- *exists()* - Return a boolean for whether a storage-pool exists by name.
- *create()* - Create a storage-pool. **Note the config in the create class method is the WHOLE json object described as 'input' in the API docs.** e.g. the 'config' key in the API docs would actually be *config.config* as passed to this method.

11.1.2 Storage-pool Object attributes

For more information about the specifics of these attributes, please see the LXD Storage Pools REST API documentation.

- *name* - the name of the storage pool
- *driver* - the driver (or type of storage pool). e.g. 'zfs' or 'btrfs', etc.
- *used_by* - which containers (by API endpoint */1.0/containers/<name>*) are using this storage-pool.
- *config* - a dictionary with some information about the storage-pool. e.g. size, source (path), volume.size, etc.
- *managed* – Boolean that indicates whether LXD manages the pool or not.

11.1.3 Storage-pool Object methods

The following methods are available on a Storage Pool object:

- *save* - save a modified storage pool. This saves the *config* attribute in it's entirety.
- *delete* - delete the storage pool.
- *put* - Change the LXD storage object with a passed parameter. The object is then synced back to the storage pool object.
- *patch* - A more fine grained patch of the object. Note that the object is then synced back after a successful patch.

Note: *raw_put* and *raw_patch* are availble (but not documented) to allow putting and patching without syncing the object back.

11.2 Storage Resources

Storage Resources are accessed from the storage pool object:

```
resources = storage_pool.resources.get()
```

Resources are read-only and there are no further methods available on them.

11.3 Storage Volumes

Storage Volumes are stored in storage pools. On the *pylxd* API they are accessed from a storage pool object:

```
storage_pool = client.storage_pools.get('pool1')
volumes = storage_pool.volumes.all()
named_volume = storage_pool.volumes.get('custom', 'vol1')
```

11.3.1 Methods available on *<storage_pool_object>.volumes*

The following methods are accessed from the *volumes* attribute on the storage pool object.

- *all* - get all the volumes on the pool.
- *get* - a get a single, type + name volume on the pool.
- *create* - create a volume on the storage pool.

Note: Note that storage volumes have a tuple of *type* and *name* to uniquely identify them. At present LXD recognises three types (but this may change), and these are *container*, *image* and *custom*. LXD uses *container* and *image* for containers and images respectively. Thus, for user applications, *custom* seems like the type of choice. Please see the LXD Storage Pools documentation for further details.

11.3.2 Methods available on the storage volume object

Once in possession of a storage volume object from the *pylxd* API, the following methods are available:

- *rename* - Rename a volume. This can also be used to migrate a volume from one pool to the other, as well as migrating to a different LXD instance.
- *put* - Put an object to the LXD server using the storage volume details and then re-sync the object.
- *patch* - Patch the object on the LXD server, and then re-sync the object back.
- *save* - after modifying the object in place, use a PUT to push those changes to the LXD server.
- *delete* - delete a storage volume object. Note that the object is, therefore, stale after this action.

Note: *raw_put* and *raw_patch* are availble (but not documented) to allow putting and patching without syncing the object back.

CHAPTER 12

Clustering

LXD supports clustering. There is only one cluster object.

12.1 Cluster object

The `Cluster` object represents the json object that is returned from *GET /1.0/cluster*.

There is also a `ClusterMember` and object that represents a cluster member at *GET /1.0/cluster/members*. Note that it should be accessed from the cluster object. For example:

```
client = pylxd.Client()
cluster = client.cluster.get()
member = cluster.members.get('node-5')
```

Note: Please see the pylxd API documentation for more information on storage pool methods and parameters. The following is a summary.

12.1.1 Cluster methods

A cluster can be queried through the following client manager methods:

- *get()* - Returns the cluster.

12.1.2 Cluster Object attributes

For more information about the specifics of these attributes, please see the LXD Cluster REST API documentation.

- *server_name* - the name of the server in the cluster
- *enabled* - if the node is enabled

- *member_config* - configuration information for new cluster members.

12.2 Cluster Members

Cluster Members are stored in a cluster. On the *pylxd* API they are accessed from a cluster object:

```
cluster = client.cluster.get()
members = cluster.members.all()
named_member = cluster.members.get('membername')
```

12.2.1 Methods available on *<cluster_object>.members*

The following methods are accessed from the *members* attribute on the cluster object.

- *all* - get all the members of the cluster.
- *get* - a get a single named member of the cluster.

12.2.2 Cluster Member Object attributes

For more information about the specifics of these attributes, please see the LXD Cluster REST API documentation.

- *server_name* - the name of the server in the cluster
- *url* - the url the lxd endpoint
- *database* - if the distributed database is replicated on this node
- *status* - if the member is off or online
- *message* - a general message

CHAPTER 13

Contributing

pyLXD development is done on Github. Pull Requests and Issues should be filed there. We try and respond to PRs and Issues within a few days.

If you would like to contribute major features or have big ideas, it's best to post at the Linux Containers disucssion forum to discuss your ideas before submitting PRs. If you use `[pylxd]` in the title, it'll make it clearer.

13.1 Adding a Feature or Fixing a Bug

The main steps are:

1. There needs to be a bug filed on the Github repository. This is also for a feature, so it's clear what is being proposed prior to somebody starting work on it.
2. The pyLXD repository must be forked on Github to the developer's own account.
3. The developer should create a personal branch, with either:
   - feature/name-of-feature
   - bug/number/descriptive-name-of-bug

   This can be done with `git checkout -b feature/name-of-feature` from the master branch.
4. Work on that branch, push to the personal GitHub repository and then create a Pull Request. It's a good idea to create the Pull Request early, particularly for features, so that it can be discussed and help sought (if needed).
5. When the Pull Request is ready it will then be merged.
6. At regular intervals the pyLXD module will be released to PyPi with the new features and bug fixes.

13.2 Requirements to merge a Pull Request (PR)

In order for a Pull Request to be merged the following criteria needs to be met:

1. All of the commits in the PR need to be signed off using the '-s' option with git commit. This is a requirement for all projects in the Github Linux Containers projects space.
2. Unit tests are required for the changes. These are in the `pylxd/tests` directory and follow the same directory structure as the module.
3. The unit test code coverage for the project shouldn't drop. This means that any lines that aren't testable (for good reasons) need to be explicitly excluded from the coverage using `# NOQA` comments.
4. If the feature/bug fix requires integration test changes, then they should be added to the `integration` directory.
5. If the feature/bug fix changes the API then the documentation in the `doc/source` directory should also be updated.
6. If the contributor is new to the project, then they should add their name/details to the `CONTRIBUTORS.rst` file in the root of the repository as part of the PR.

Once these requirements are met, the change will be merged to the repository. At this point, the contributor should then delete their private branch.

13.3 Code standards

pyLXD follows PEP 8 as closely as practical. To check your compliance, use the *pep8* tox target:

```
tox -e pep8
```

Note: if this fails then the code will not be merged. If there is a good reason for a PEP8 non-conformance, then a `# NOQA` comment should be added to the relevant line(s).

13.4 Testing

Testing pyLXD is in 3 parts:

1. Conformance with PEP 8, using the `tox -e pep8` command.
2. Unit tests using `tox -e py27` and `tox -e py3`.
3. Integration tests using the `run_integration_tests` script in the root of the repository.

Note: all of the tests can be run by just using the `tox` command on it's own, with the exception of the integration tests. These are not automatically run as they require a working LXD environment.

All of the commands use the Tox automation project to run tests in a sandboxed environment. On Ubuntu this is installed using:

```
sudo apt install python-tox
```

13.4.1 Unit Testing

pyLXD tries to follow best practices when it comes to testing. PRs are gated by Travis CI and CodeCov. It's best to submit tests with new changes, as your patch is unlikely to be accepted without them.

To run the tests, you should use Tox:

```
tox
```

13.4.2 Integration Testing

Integration testing requires a running LXD system. At present this is not performed by the CI system, although this is intended at some point in the future. Integration testing *should* be performed prior to merging a PR.

Currently, there are two variants of the script to run integration tests:

1. `run_integration_tests-16-04`
2. `run_integration_tests-18-04`

The default is `run_integration_tests-18-04`, which is symlinked to `run_integration_tests`. This is because the default is to test on Ubuntu Bionic, with Ubuntu Xenial (16.04) for maintenance purposes.

Note: A script to automate running the integration tests needs to be added.

Some hints on how to run the integration tests:

1. On Ubuntu it's probably easiest to use the Multipass snap.
2. Launch an LTS instance using `multipass launch -n foo`
3. Shell into the instance: `multipass exec foo -- bash`
4. Install tox and python2.7: `sudo apt install python-tox python-2.7`
5. Clone the branch from the PR (or otherwise copy the repo into the machine)
6. Configure LXD using `lxd init` – follow the prompts provided.
7. Run the integration tests.

CHAPTER 14

API documentation

14.1 Client

class pylxd.client.**Client**(*endpoint=None*, *version='1.0'*, *cert=None*, *verify=True*, *timeout=None*)

Client class for LXD REST API.

This client wraps all the functionality required to interact with LXD, and is meant to be the sole entry point.

instances
Instance of `Client.Instances`:

containers
Instance of `Client.Containers`:

virtual_machines
Instance of `Client.VirtualMachines`:

images
Instance of `Client.Images`.

operations
Instance of `Client.Operations`.

profiles
Instance of `Client.Profiles`.

api
This attribute provides tree traversal syntax to LXD's REST API for lower-level interaction.

Use the name of the url part as attribute or item of an api object to create another api object appended with the new url part name, ie:

```
>>> api = Client().api
# /
>>> response = api.get()
# Check status code and response
```

(continues on next page)

(continued from previous page)

```
>>> print response.status_code, response.json()
# /instances/test/
>>> print api.instances['test'].get().json()
```

assert_has_api_extension(*name*)
Asserts that the *name* api_extension exists. If not, then is raises the LXDAPIExtensionNotAvailable error.

Parameters **name** (*str*) – the api_extension to test for

Returns None

Raises pylxd.exceptions.LXDAPIExtensionNotAvailable

events(*websocket_client=None*, *event_types=None*)
Get a websocket client for getting events.

/events is a websocket url, and so must be handled differently than most other LXD API endpoints. This method returns a client that can be interacted with like any regular python socket.

An optional *websocket_client* parameter can be specified for implementation-specific handling of events as they occur.

Parameters

- **websocket_client** (*ws4py.client import WebSocketBaseClient*) – Optional websocket client can be specified for implementation-specific handling of events as they occur.
- **event_types** (*Set[EventType]*) – Optional set of event types to propagate. Omit this argument or specify {EventTypes.All} to receive all events.

Returns instance of the websocket client

Return type Option[_WebsocketClient(), **:param:'websocket_client'**]

has_api_extension(*name*)
Return True if the *name* api extension exists.

Parameters **name** (*str*) – the api_extension to look for.

Returns True if extension exists

Return type bool

14.2 Exceptions

class pylxd.exceptions.**LXDAPIException**(*response*)
A generic exception for representing unexpected LXD API responses.

LXD API responses are clearly documented, and are either a standard return value, and background operation, or an error. This exception is raised on an error case, or when the response status code is not expected for the response.

This exception should *only* be raised in cases where the LXD REST API has returned something unexpected.

class pylxd.exceptions.**NotFound**(*response*)
An exception raised when an object is not found.

class pylxd.exceptions.**ClientConnectionFailed**
An exception raised when the Client connection fails.

14.3 Certificate

class pylxd.models.**Certificate**(*client*, ***kwargs*)
A LXD certificate.

classmethod all(*client*)
Get all certificates.

classmethod create(*client*, *password*, *cert_data*)
Create a new certificate.

classmethod get(*client*, *fingerprint*)
Get a certificate by fingerprint.

14.4 Container

class pylxd.models.**Container**(**args*, ***kwargs*)

class pylxd.models.**Snapshot**(*client*, ***kwargs*)
A instance snapshot.

publish(*public=False*, *wait=False*)
Publish a snapshot as an image.

If wait=True, an Image is returned.

This functionality is currently broken in LXD. Please see https://github.com/lxc/lxd/issues/2201 - The implementation here is mostly a guess. Once that bug is fixed, we can verify that this works, or file a bug to fix it appropriately.

rename(*new_name*, *wait=False*)
Rename a snapshot.

restore(*wait=False*)
Restore this snapshot.

Attempts to restore a instance using this snapshot. The instance should be stopped, but the method does not enforce this constraint, so an LXDAPIException may be raised if this method fails.

Parameters **wait** (*boolean*) – wait until the operation is completed.

Raises LXDAPIException if the the operation fails.

Returns the original response from the restore operation (not the operation result)

Return type requests.Response

14.5 Image

class pylxd.models.**Image**(*client*, ***kwargs*)
A LXD Image.

add_alias(*name*, *description*)
Add an alias to the image.

classmethod all(*client*)
Get all images.

copy(*new_client*, *public=None*, *auto_update=None*, *wait=False*)
Copy an image to a another LXD.

Destination host information is contained in the client connection passed in.

classmethod create(*client*, *image_data*, *metadata=None*, *public=False*, *wait=True*)
Create an image.

If metadata is provided, a multipart form data request is formed to push metadata and image together in a single request. The metadata must be a tar achive.

wait parameter is now ignored, as the image fingerprint cannot be reliably determined consistently until after the image is indexed.

classmethod create_from_simplestreams(*client*, *server*, *alias*, *public=False*, *auto_update=False*)
Copy an image from simplestreams.

classmethod create_from_url(*client*, *url*, *public=False*, *auto_update=False*)
Copy an image from an url.

delete_alias(*name*)
Delete an alias from the image.

classmethod exists(*client*, *fingerprint*, *alias=False*)
Determine whether an image exists.

If *alias* is True, look up the image by its alias, rather than its fingerprint.

export()
Export the image.

Because the image itself may be quite large, we stream the download in 1kb chunks, and write it to a temporary file on disk. Once that file is closed, it is deleted from the disk.

classmethod get(*client*, *fingerprint*)
Get an image.

classmethod get_by_alias(*client*, *alias*)
Get an image by its alias.

14.6 Network

class pylxd.models.**Network**(*client*, ***kwargs*)
Model representing a LXD network.

classmethod all(*client*)
Get all networks.

Parameters **client** (*Client*) – client instance

Return type list[*Network*]

classmethod create(*client*, *name*, *description=None*, *type=None*, *config=None*)
Create a network.

Parameters

- **client** (*Client*) – client instance
- **name** (*str*) – name of the network
- **description** (*str*) – description of the network

- **type** (*str*) – type of the network
- **config** (*dict*) – additional configuration

classmethod exists(*client*, *name*)
Determine whether network with provided name exists.

Parameters

- **client** (*Client*) – client instance
- **name** (*str*) – name of the network

Returns *True* if network exists, *False* otherwise

Return type bool

classmethod get(*client*, *name*)
Get a network by name.

Parameters

- **client** (*Client*) – client instance
- **name** (*str*) – name of the network

Returns network instance (if exists)

Return type *Network*

Raises *NotFound* if network does not exist

rename(*new_name*)
Rename a network.

Parameters **new_name** (*str*) – new name of the network

Returns Renamed network instance

Return type *Network*

save(**args*, ***kwargs*)
Save data to the server.

This method should write the new data to the server via marshalling. It should be a no-op when the object is not dirty, to prevent needless I/O.

14.7 Operation

class pylxd.models.**Operation**(***kwargs*)
An LXD operation.

If the LXD server sends attributes that this version of pylxd is unaware of then a warning is printed. By default the warning is issued ONCE and then supressed for every subsequent attempted setting. The warnings can be completely suppressed by setting the environment variable PYLXD_WARNINGS to 'none', or always displayed by setting the PYLXD_WARNINGS variable to 'always'.

classmethod get(*client*, *operation_id*)
Get an operation.

wait()
Wait for the operation to complete and return.

classmethod wait_for_operation(*client*, *operation_id*)
Get an operation and wait for it to complete.

14.8 Profile

class pylxd.models.**Profile**(*client*, ***kwargs*)
A LXD profile.

classmethod all(*client*)
Get all profiles.

classmethod create(*client*, *name*, *config=None*, *devices=None*)
Create a profile.

classmethod exists(*client*, *name*)
Determine whether a profile exists.

classmethod get(*client*, *name*)
Get a profile.

rename(*new_name*)
Rename the profile.

14.9 Storage Pool

class pylxd.models.**StoragePool**(**args*, ***kwargs*)
An LXD storage_pool.

This corresponds to the LXD endpoint at /1.0/storage-pools

api_extension: 'storage'

classmethod all(*client*)
Get all storage_pools.

Implements GET /1.0/storage-pools

Note that the returned list is 'sparse' in that only the name of the pool is populated. If any of the attributes are used, then the *sync* function is called to populate the object fully.

Parameters **client** (*pylxd.client.Client*) – The pylxd client object

Returns a storage pool if successful, raises NotFound if not found

Return type [pylxd.models.storage_pool.StoragePool]

Raises pylxd.exceptions.LXDAPIExtensionNotAvailable if the 'storage' api extension is missing.

api
Provides an object with the endpoint:

/1.0/storage-pools/<self.name>

Used internally to construct endpoints.

Returns an API node with the named endpoint

Return type pylxd.client._APINode

classmethod create(*client*, *definition*)
Create a storage_pool from config.

Implements POST /1.0/storage-pools

The *definition* parameter defines what the storage pool will be. An example config for the zfs driver is:

{

"config": { "size": "10GB"

}, "driver": "zfs", "name": "pool1"

}

Note that **all** fields in the *definition* parameter are strings.

For further details on the storage pool types see: https://lxd.readthedocs.io/en/latest/storage/

The function returns the a *StoragePool* instance, if it is successfully created, otherwise an Exception is raised.

Parameters

- **client** (*pylxd.client.Client*) – The pylxd client object
- **definition** (*dict*) – the fields to pass to the LXD API endpoint

Returns a storage pool if successful, raises NotFound if not found

Return type pylxd.models.storage_pool.StoragePool

Raises pylxd.exceptions.LXDAPIExtensionNotAvailable if the 'storage' api extension is missing.

Raises *pylxd.exceptions.LXDAPIException* if the storage pool couldn't be created.

delete()
Delete the storage pool.

Implements DELETE /1.0/storage-pools/<self.name>

Deleting a storage pool may fail if it is being used. See the LXD documentation for further details.

Raises *pylxd.exceptions.LXDAPIException* if the storage pool can't be deleted.

classmethod exists(*client*, *name*)
Determine whether a storage pool exists.

A convenience method to determine a pool exists. However, it is better to try to fetch it and catch the *pylxd.exceptions.NotFound* exception, as otherwise the calling code is like to fetch the pool twice. Only use this if the calling code doesn't *need* the actual storage pool information.

Parameters

- **client** (*pylxd.client.Client*) – The pylxd client object
- **name** (*str*) – the name of the storage pool to get

Returns True if the storage pool exists, False if it doesn't.

Return type bool

Raises pylxd.exceptions.LXDAPIExtensionNotAvailable if the 'storage' api extension is missing.

classmethod get(*client*, *name*)
Get a storage_pool by name.

Implements GET /1.0/storage-pools/<name>

Parameters

- **client** (*pylxd.client.Client*) – The pylxd client object
- **name** (*str*) – the name of the storage pool to get

Returns a storage pool if successful, raises NotFound if not found

Return type pylxd.models.storage_pool.StoragePool

Raises *pylxd.exceptions.NotFound*

Raises pylxd.exceptions.LXDAPIExtensionNotAvailable if the 'storage' api extension is missing.

patch(*patch_object*, *wait=False*)
Patch the storage pool.

Implements PATCH /1.0/storage-pools/<self.name>

Patching the object allows for more fine grained changes to the config. The object is refreshed if the PATCH is successful. If this is *not* required, then use the client api directly.

Parameters

- **patch_object** (*dict*) – A dictionary. The most useful key will be the *config* key.
- **wait** (*bool*) – Whether to wait for async operations to complete.

Raises *pylxd.exceptions.LXDAPIException* if the storage pool can't be modified.

put(*put_object*, *wait=False*)
Put the storage pool.

Implements PUT /1.0/storage-pools/<self.name>

Putting to a storage pool may fail if the new configuration is incompatible with the pool. See the LXD documentation for further details.

Note that the object is refreshed with a *sync* if the PUT is successful. If this is *not* desired, then the raw API on the client should be used.

Parameters

- **put_object** (*dict*) – A dictionary. The most useful key will be the *config* key.
- **wait** (*bool*) – Whether to wait for async operations to complete.

Raises *pylxd.exceptions.LXDAPIException* if the storage pool can't be modified.

save(*wait=False*)
Save the model using PUT back to the LXD server.

Implements PUT /1.0/storage-pools/<self.name> *automagically*

The fields affected are: *description* and *config*. Note that they are replaced in their *entirety*. If finer grained control is required, please use the patch() method directly.

Updating a storage pool may fail if the config is not acceptable to LXD. An *LXDAPIException* will be generated in that case.

Raises *pylxd.exceptions.LXDAPIException* if the storage pool can't be deleted.

14.10 Cluster

class pylxd.models.**Cluster**(**args*, ***kwargs*)
An LXD Cluster.

classmethod get(*client*, **args*)
Get cluster details

class pylxd.models.**ClusterMember**(*client*, ***kwargs*)
A LXD cluster member.

classmethod all(*client*, **args*)
Get all cluster members.

classmethod get(*client*, *server_name*)
Get a cluster member by name.

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Index

R

S

V

W