# pylxd Documentation

**Canonical Ltd**

**May 24, 2019**

# 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.

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.

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.
- *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.
- *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.

## 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.

```
>>> config = {'name': 'my-container', 'source': {'type': 'image', 'alias': 'ubuntu/
↪trusty'}}
>>> 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 *save* 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'}
```

## 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.

```
>>> 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=False)* - 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').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 (see *network_extension_available()*)

## 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* - config options for containers
- *description* - The description of the profile
- *devices* - device options for containers
- *name* - The name of the profile
- *used_by* - A list of 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 = profile.config.update({'security.nesting': 'true'})
>>> profile.update()
```

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 network, by its name.
- *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.

*Storage Pool* 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.

## 11.1 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.

## 11.2 Storage-pool attributes

For more information about the specifics of these attributes, please see the LXD 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 string (json encoded) with some information about the storage-pool. e.g. size, source (path), volume.size, etc.

## 11.3 Storage-pool methods

The are no storage pool methods defined yet.

CHAPTER 12

# 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 large features or have big ideas, it's best to post on to the lxc-users list to discuss your ideas before submitting PRs.

## 12.1 Code standards

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

```
tox -epep8
```

## 12.2 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 can use nose:

```
nosetests pylxd
```

. . . or, alternatively, you can use *tox* (with the added bonus that it will test python 2.7, python 3, and pypy, as well as run pep8). This is the way that Travis will test, so it's recommended that you run this at least once before submitting a Pull Request.

# CHAPTER 13

# API documentation

## 13.1 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.

**containers**
Instance of `Client.Containers`:

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

**events**(*websocket_client=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.

## 13.2 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.

## 13.3 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.

## 13.4 Container

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

This class is not intended to be used directly, but rather to be used via *Client.containers.create*.

**class FilesManager**(*client*, *container*)
A pseudo-manager for namespacing file operations.

**delete_available**()
File deletion is an extension API and may not be available. https://github.com/lxc/lxd/blob/master/doc/api-extensions.md#file_delete

**put**(*filepath*, *data*, *mode=None*, *uid=None*, *gid=None*)
Push a file to the container.

This pushes a single file to the containers file system named by the *filepath*.

**Parameters**
- **filepath** (*str*) – The path in the container to to store the data in.
- **data** (*bytes or str*) – The data to store in the file.
- **mode** (*Union[oct, int, str]*) – The unit mode to store the file with. The default of None stores the file with the current mask of 0700, which is the lxd default.
- **uid** (*int*) – The uid to use inside the container. Default of None results in 0 (root).
- **gid** (*int*) – The gid to use inside the container. Default of None results in 0 (root).

**Raises** LXDAPIException if something goes wrong

**recursive_put**(*src*, *dst*, *mode=None*, *uid=None*, *gid=None*)
Recursively push directory to the container.

Recursively pushes directory to the containers named by the *dst*

**Parameters**
- **src** (*str*) – The source path of directory to copy.
- **dst** (*str*) – The destination path in the container of directory to copy
- **mode** (*Union[oct, int, str]*) – The unit mode to store the file with. The default of None stores the file with the current mask of 0700, which is the lxd default.
- **uid** (*int*) – The uid to use inside the container. Default of None results in 0 (root).
- **gid** (*int*) – The gid to use inside the container. Default of None results in 0 (root).

**Raises** NotADirectoryError if src is not a directory

**Raises** LXDAPIException if an error occurs

**classmethod all**(*client*)
Get all containers.

Containers returned from this method will only have the name set, as that is the only property returned from LXD. If more information is needed, *Container.sync* is the method call that should be used.

**classmethod create**(*client*, *config*, *wait=False*)
Create a new container config.

**execute**(*commands*, *environment={}*, *encoding=None*, *decode=True*)
Execute a command on the container.

In pylxd 2.2, this method will be renamed *execute* and the existing *execute* method removed.

**Parameters**
- **commands** (*[str]*) – The command and arguments as a list of strings
- **environment** (*{str: str}*) – The environment variables to pass with the command
- **encoding** (*str*) – The encoding to use for stdout/stderr if the param decode is True. If encoding is None, then no override is performed and whatever the existing encoding from LXD is used.
- **decode** (*bool*) – Whether to decode the stdout/stderr or just return the raw buffers.

**Raises** **ValueError** – if the ws4py library is not installed.

**Returns** The return value, stdout and stdin

**Return type** _ContainerExecuteResult() namedtuple

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

**freeze**(*timeout=30*, *force=True*, *wait=False*)
Freeze the container.

**generate_migration_data**()
Generate the migration data.

This method can be used to handle migrations where the client connection uses the local unix socket. For more information on migration, see *Container.migrate*.

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

**migrate**(*new_client*, *wait=False*)
Migrate a container.

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

If the container is running, it either must be shut down first or criu must be installed on the source and destination machines.

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

The container must be stopped in order publish it as an image. This method does not enforce that constraint, so a LXDAPIException may be raised if this method is called on a running container.

If wait=True, an Image is returned.

**rename**(*name*, *wait=False*)
Rename a container.

**restart**(*timeout=30*, *force=True*, *wait=False*)
Restart the container.

**start**(*timeout=30*, *force=True*, *wait=False*)
Start the container.

**stop**(*timeout=30*, *force=True*, *wait=False*)
Stop the container.

**unfreeze**(*timeout=30*, *force=True*, *wait=False*)
Unfreeze the container.

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

## 13.5 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.

## 13.6 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

**static network_extension_available**(*client*)
Network operations is an extension API and may not be available.

https://github.com/lxc/lxd/blob/master/doc/api-extensions.md#network

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

**Returns** *True* if network API extension is available, *False* otherwise.

**Return type** *bool*

**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.

## 13.7 Operation

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

**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.

## 13.8 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.

## 13.9 Storage Pool

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

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

**classmethod all**(*client*)
Get all storage_pools.

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

**delete**()
Delete is not available for storage_pools.

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

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

**save**(*wait=False*)
Save is not available for storage_pools.

CHAPTER 14

# Indices and tables

- genindex
- modindex
- search

# Index

## O



## P



## R



## S



## U



## W