pylxd Documentation

Canonical Ltd

Nov 13, 2023

CONTENTS

Contents:

CHAPTER

ONE

INSTALLATION

Install pylxd using pip:

```
pip install pylxd
```

CHAPTER

TWO

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 and a PEM file containing the LXD's certificate to the *verify* argument:

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

In the case where the certificate is self-signed (LXD's default), you may opt to disable the TLS fingerprint verification with *verify=False*. As this disables an important security feature, doing so is strongly discouraged. The client filesystem will be searched for potential certificate to use for TLS verification.

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

If LXD is configured to use projects and you would like to interact with a specific one, you can specify its name as the *project* argument.

```
>>> from pylxd import Client
>>> client = Client(project='my-project')
```

Note: omitting the *project* argument will connect the API client to the default project of the LXD instance.

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

THREE

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 -x509 -newkey rsa:2048 -keyout lxd.key -nodes -out lxd.crt -subj "/CN=lxd.
↪local"
```

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

FOUR

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:

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

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

```
>>> types = set([EventType.Lifecycle])
>>> ws_client = client.events(event_types=types)
```

CHAPTER

FIVE

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

SIX

INSTANCES

Instance objects are the core of LXD. They are the result of supporting the management of virtual machines. Instances can be created, updated, and deleted. Most of the methods for operating on the instance itself are asynchronous, but many of the methods for getting information about the instance are synchronous.

Instead of the *instances* model, separate *containers* and *virtual_machines* exist which only return instances of the specific type.

6.1 Manager methods

Instances can be queried through the following client manager methods:

- *exists(name)* - Returns *boolean* indicating if the instance exists.
- *all()* - Retrieve all instances.
- *get()* - Get a specific instance, by its name.
- *create(config, wait=False, target='lxd-cluster-member')* - Create a new instance. - This method requires the instance 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*. - Optionally, the target node can be specified for LXD clusters.

6.2 Instance attributes

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

- *architecture* - The instance architecture.
- *config* - The instance config
- *created_at* - The time the instance was created
- *devices* - The devices for the instance
- *ephemeral* - Whether the instance is ephemeral
- *expanded_config* - An expanded version of the config
- *expanded_devices* - An expanded version of devices
- *name* - (Read only) The name of the instance. This attribute serves as the primary identifier of an instance
- *description* - A description given to the instance
- *profiles* - A list of profiles applied to the instance

- *status* - (Read only) A string representing the status of the instance
- *last_used_at* - (Read only) when the instance was last used
- *location* - (Read only) the host of the instance in a cluster
- *type* - (Read only) whether a container (default) or virtual-machine
- *status_code* - (Read only) A LXD status code of the instance
- *stateful* - (Read only) Whether the instance is stateful

6.3 Instance methods

- *rename* - Rename an instance. Because *name* is the key, it cannot be renamed by simply changing the name of the instance 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 instance's configuration
- *state* - Get the expanded state of the instance.
- *start* - Start the instance
- *stop* - Stop the instance
- *restart* - Restart the instance
- *freeze* - Suspend the instance
- *unfreeze* - Resume the instance
- *execute* - Execute a command on the instance. 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 instance. It will return an url to an interactive websocket and the execution only starts after a client connected to the websocket.
- *migrate* - Migrate the instance. The first argument is a client connection to the destination server. This call is asynchronous, so `wait=True` is optional. The instance on the new client is returned. If `live=True` is passed to the function call, then the instance is live migrated (see the LXD documentation for further details).
- *publish* - Publish the instance as an image. Note the instance 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 instance by its name…

```
>>> client.instances.get('my-instance')
<instance.Instance at 0x7f95d8af72b0>
```

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

```
>>> client.instances.all()
[<instance.Instance at 0x7f95d8af72b0>,]
```

In order to create a new `Instance`, an instance 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 instance to be created before the command returns, you'll pass *wait=True* as well.

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

If you were to use an actual local image source, you would be able to operate on the instance: starting, stopping, freezing, deleting, etc. You could also customize the instance's config (limits and etc). Note that depends on having a local image with the alias *focal*. See the next example for using a remote image.

```
>>> config = {'name': 'my-instance', 'source': {'type': 'image', 'alias': 'focal'},
↪'config': {'limits.cpu': '2'}}
>>> instance = client.instances.create(config, wait=True)
>>> instance.start()
>>> instance.freeze()
>>> instance.delete()
```

Config line with a remote image source (daily build of the latest Ubuntu LTS) and a single profile named *profilename*.

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

To modify instance's configuration method ` ` should be called after `Instance` attributes changes.

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

To get state information such as a network address.

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

To migrate an instance 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
```

(continues on next page)

(continued from previous page)

```
↪'),verify=False)
cont = client_source.instances.get('testm')
cont.migrate(client_destination,wait=True)
```

This will migrate the instance from source server to destination server

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

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

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

```
>>> res = instance.raw_interactive_execute(['/bin/bash'])
>>> res
{
    "name": "instance-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 Instance Snapshots

Each instance 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 <instance-name>/<snapshot-name>, but the manager allows us to use our own namespacing).

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

- *rename* - rename a snapshot
- *publish* - create an image from a snapshot.
- *restore* - restore the instance to this snapshot.

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

(continues on next page)

(continued from previous page)

```
>>> snapshot.name
'my-backup'
```

6.6 Instance files

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

- *put* - push a file into the instance.
- *mk_dir* - create an empty directory on the instance.
- *recursive_put* - recursively push a directory to the instance.
- *get* - get a file from the instance.
- *recursive_get* - recursively pull a directory from the instance.
- *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 instance.

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

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

CHAPTER

SEVEN

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.
- *delete* - Deletes the image.

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'
```

Finally, delete an image. As this is an asynchonous operation, you may also want to *wait=True*.

```
>>> image = client.images.get(
...     'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855')
>>> image.delete(wait=True)
```

CHAPTER

EIGHT

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 Network 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={})
```

CHAPTER

NINE

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

TEN

PROJECTS

LXD supports projects as a way to split your LXD server. Each *Project* holds its own set of instances and may also have its own images and profiles.

10.1 Manager methods

Projects can be queried through the following client manager methods:

- *all()* - Retrieve all projects.
- *exists()* - See if a project with a name exists. Returns *boolean*.
- *get()* - Get a specific project, by its name.
- *create()* - Create a new project. The *name* of the project is required. *description* is an optional string with a description of the project. The *config* dictionary is also optional, the scope of which is documented in the LXD project documentation.

10.2 Project attributes

- `name` - (str) name of the project.
- `description` - (str) The description of the project.
- `config` - (dict) config options for the project.
- `used_by` - (list) images, instances, networks, and profiles using this project.

10.3 Project methods

- *rename()* - Rename the project.
- `save()` - save a project. This uses the PUT HTTP method and not the PATCH.
- `delete()` - deletes a project.

10.4 Examples

Project operations follow the same manager-style as Containers and Images. Projects are keyed on a unique name.

```
>>> project = client.projects.get('my-project')
>>> project
<project.Project at 0x7f599e129a60>
```

The project can then be modified and saved.

```
>>> project.config['limits.instances'] = '4'
>>> project.description = "The four horsemen of the apococalypse"
>>> project.save()
```

To create a new project, use *create* with a name, optional *description* string and *config* dictionary.

```
>>> project = client.projects.create(
...     'a-project', description="New project", config={'limits.instances': '10'})
```

CHAPTER

ELEVEN

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.

11.1 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.

11.2 Operation object methods

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

CHAPTER

TWELVE

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.

12.1 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.

12.1.1 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.

12.1.2 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.

12.1.3 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.

12.2 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.

12.3 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')
```

12.3.1 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.

12.3.2 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

THIRTEEN

CLUSTERING

LXD supports clustering. There is only one cluster object.

13.1 Cluster object

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

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

13.1.1 Cluster methods

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

- *get()* - Returns the cluster.
- *enable(server_name)* - Enable clustering.

13.1.2 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.

13.2 Cluster Members objects

The `ClusterMember` object represents the json object that is returned from *GET /1.0/cluster/members/<name>*. For example:

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

13.2.1 Methods available on <*clustermember_object*>

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

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

13.2.2 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

FOURTEEN

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 LXD category in Ubuntu's Discourse to discuss your ideas before submitting PRs. If you use `[pylxd]` in the title, it'll make it clearer.

14.1 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 main 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.

14.2 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.
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.

14.3 Code standards

pyLXD formats code with Black and isort. Verify the formatting with:

```
tox -e lint
```

If it fails, you can reformat the code with:

```
tox -e format
```

14.4 Testing

Testing pyLXD is in 3 parts:

1. Conformance with Black and isort, using the `tox -e lint` command.
2. Unit tests using `tox -e py` or `tox -e coverage`.
3. Integration tests using the `tox -e integration-in-lxd`.

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.

14.4.1 Unit Testing

pyLXD tries to follow best practices when it comes to testing. PRs are gated by GitHub Actions 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
```

14.4.2 Integration Testing

Integration testing requires a running LXD system. They can be tested locally in LXD container with nesting support; `tox -e integration-in-lxd`.

CHAPTER

FIFTEEN

API DOCUMENTATION

15.1 Client

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

A *models.Instance*.

containers

A *models.Container*.

virtual_machines

A *models.VirtualMachine*.

images

A *models.Image*.

operations

A *models.Operation*.

profiles

A *models.Profile*.

projects

A *models.Project*.

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()
# /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(), *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

15.2 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.

15.3 Certificate

class pylxd.models.**Certificate**(*client*, ***kwargs*)

A LXD certificate.

classmethod all(*client*)

Get all certificates.

classmethod create(*client*, *password*, *cert_data*, *cert_type='client'*, *name=''*, *projects=None*, *restricted=False*)

Create a new certificate.

classmethod create_token(*client*, *name=''*, *projects=None*, *restricted=False*)

Create a new token.

classmethod get(*client*, *fingerprint*)

Get a certificate by fingerprint.

15.4 Container

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

An LXD Instance.

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

class FilesManager(*instance*)

A pseudo-manager for namespacing file operations.

delete_available()

File deletion is an extension API and may not be available. https://documentation.ubuntu.com/lxd/en/latest/api-extensions/#file-delete

mk_dir(*path*, *mode=None*, *uid=None*, *gid=None*)

Creates an empty directory on the container. This pushes an empty directory to the containers file system named by the *filepath*.

Parameters

- **path** (*str*) – The path in the container to to store the data in.
- **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

put(*filepath*, *data*, *mode=None*, *uid=None*, *gid=None*)

Push a file to the instance.

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

Parameters

- **filepath** (*str*) – The path in the instance 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 instance. Default of None results in 0 (root).
- **gid** (*int*) – The gid to use inside the instance. Default of None results in 0 (root).

Raises

LXDAPIException if something goes wrong

recursive_get(*remote_path*, *local_path*)

Recursively pulls a directory from the container. Pulls the directory named *remote_path* from the container and creates a local folder named *local_path* with the content of *remote_path*. If *remote_path* is a file, it will be copied to *local_path*.

Parameters

- **remote_path** (*str*) – The directory path on the container.
- **local_path** (*str*) – The path at which the directory will be stored.

Raises

LXDAPIException if an error occurs

recursive_put(*src*, *dst*, *mode=None*, *uid=None*, *gid=None*)

Recursively push directory to the instance.

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

Parameters

- **src** (*str*) – The source path of directory to copy.
- **dst** (*str*) – The destination path in the instance 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 instance. Default of None results in 0 (root).
- **gid** (*int*) – The gid to use inside the instance. Default of None results in 0 (root).

Raises

NotADirectoryError if src is not a directory

Raises

LXDAPIException if an error occurs

classmethod all(*client*, *recursion=0*)

Get all instances.

This method returns an Instance array. If recursion is unset, only the name of each instance will be set and *Instance.sync* can be used to return more information. If recursion is between 1-2 this method will pre-fetch additional instance attributes for all instances in the array.

classmethod create(*client*, *config*, *wait=False*, *target=None*)

Create a new instance config.

Parameters

- **client** (Client) – client instance
- **config** (*dict*) – The configuration for the new instance.
- **wait** (*bool*) – Whether to wait for async operations to complete.
- **target** (*str*) – If in cluster mode, the target member.

Raises
LXDAPIException – if something goes wrong.

Returns
an instance if successful

Return type
Instance

execute(*commands*, *environment=None*, *encoding=None*, *decode=True*, *stdin_payload=None*, *stdin_encoding='utf-8'*, *stdout_handler=None*, *stderr_handler=None*, *user=None*, *group=None*, *cwd=None*)

Execute a command on the instance. stdout and stderr are buffered if no handler is given.

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.
- **stdin_payload** (*Can be a file, string, bytearray, generator or ws4py Message object*) – Payload to pass via stdin
- **stdin_encoding** – Encoding to pass text to stdin (default utf-8)
- **stdout_handler** (*Callable[[str], None]*) – Callable than receive as first parameter each message received via stdout
- **stderr_handler** (*Callable[[str], None]*) – Callable than receive as first parameter each message received via stderr
- **user** (*int*) – User to run the command as
- **group** (*int*) – Group to run the command as
- **cwd** (*str*) – Current working directory

Returns
A tuple of *(exit_code, stdout, stderr)*

Return type
_InstanceExecuteResult() namedtuple

classmethod exists(*client*, *name*)

Determine whether a instance exists.

freeze(*timeout=30*, *force=True*, *wait=False*)

Freeze the instance.

generate_migration_data(*live=False*)

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 *Instance.migrate*.

Parameters
live (*bool*) – Whether to include "live": "true" in the migration

Raises
LXDAPIException if the request to migrate fails

Returns
dictionary of migration data suitable to send to an new client to complete a migration.

Return type
Dict[str, ANY]

classmethod get(*client*, *name*)

Get a instance by name.

migrate(*new_client*, *live=False*, *wait=False*)

Migrate a instance.

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

If the *live* param is True, then a live migration is attempted, otherwise a non live one is running.

If the instance is running for live migration, it either must be shut down first or criu must be installed on the source and destination machines and the *live* param should be True.

Parameters

- **new_client** (*pylxd.client.Client*) – the pylxd client connection to migrate the instance to.
- **live** (*bool*) – whether to perform a live migration
- **wait** (*bool*) – if True, wait for the migration to complete

Raises
LXDAPIException if any of the API calls fail.

Raises
ValueError if source of target are local connections

Returns
the response from LXD of the new instance (the target of the migration and not the operation if waited on.)

Return type
requests.Response

publish(*public=False*, *wait=False*)

Publish a instance as an image.

The instance 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 instance.

If wait=True, an Image is returned.

raw_interactive_execute(*commands*, *environment=None*, *user=None*, *group=None*, *cwd=None*)

Execute a command on the instance interactively and returns urls to websockets. The urls contain a secret uuid, and can be accesses without further authentication. The caller has to open and manage the websockets themselves.

Parameters

- **commands** (*[str]*) – The command and arguments as a list of strings (most likely a shell)
- **environment** (*{str: str}*) – The environment variables to pass with the command
- **user** (*int*) – User to run the command as

- **group** (*int*) – Group to run the command as
- **cwd** (*str*) – Current working directory

Returns
Two urls to an interactive websocket and a control socket

Return type
{'ws':str,'control':str}

rename(*name*, *wait=False*)

Rename an instance.

restart(*timeout=30*, *force=True*, *wait=False*)

Restart the instance.

restore_snapshot(*snapshot_name*, *wait=False*)

Restore a snapshot using its name.

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

Parameters

- **snapshot_name** (*str*) – the name of the snapshot to restore from
- **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`

start(*timeout=30*, *force=True*, *wait=False*)

Start the instance.

stop(*timeout=30*, *force=True*, *wait=False*)

Stop the instance.

unfreeze(*timeout=30*, *force=True*, *wait=False*)

Unfreeze the instance.

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

Flavour of *models.Instance* for containers.

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.

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

15.5 Virtual Machine

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

Flavour of *models.Instance* for VMs.

15.6 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.

15.7 Network

class pylxd.models.**Network**(**args*, ***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.

state()

Get network state.

15.8 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.

15.9 Profile

class pylxd.models.**Profile**(*client*, ***kwargs*)

A LXD profile.

classmethod all(*client*)

Get all profiles.

classmethod create(*client*, *name*, *config=None*, *devices=None*, *description=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.

15.10 Project

class pylxd.models.**Project**(*client*, ***kwargs*)

A LXD project.

This corresponds to the LXD endpoint at /1.0/projects.

api_extension: 'projects'

classmethod all(*client*)

Get all projects.

classmethod create(*client*, *name*, *description=None*, *config=None*)

Create a project.

classmethod exists(*client*, *name*)

Determine whether a project exists.

classmethod get(*client*, *name*)

Get a project.

rename(*new_name*)

Rename the project.

15.11 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.

property 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://documentation.ubuntu.com/lxd/en/latest/explanation/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.

15.12 Cluster

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

An LXD Cluster.

classmethod enable(*client*, *server_name*)

Enable clustering on a single non-clustered LXD server.

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

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX