
Idapper Documentation

Release 1.0.3

UMIACS Staff

Jun 09, 2020

Contents

1	Contents:	3
2	Indices and tables	11
	Python Module Index	13
	Index	15

Ldapper is a simple ORM for interacting with an LDAP. It makes it straightforward to create models to perform all your CRUD operations.

1.1 Installation

1.1.1 Via Python Package

Install the latest version from PyPI:

```
pip install ldapper
```

1.1.2 Via Source

The project is hosted at <https://github.com/umiacs/ldapper> and can be installed using git:

```
git clone https://github.com/umiacs/ldapper.git
cd ldapper
python setup.py install
```

1.2 Configuration

To get started, you'll want to create a connection class. It will contain the settings necessary to connect to your LDAP.

```
from ldapper.connection import BaseConnection

class Connection(BaseConnection):
    BASE_DN = 'dc=example,dc=com'
    URI = 'ldaps://ldap.example.com'
```

You'll go on to create a class that all of your models can inherit from in order to pick up on the connection settings that you want them to be backed against.

Creating this class is not strictly necessary, but it is convenient if all of your models are going to use the same connection.

```
from ldapper.ldapnode import LDAPNode

class BaseModel(LDAPNode):
    connection = Connection
```

Create your first model:

```
from ldapper.fields import ListField, StringField

class Person(BaseModel):
    uid = StringField('uid', primary=True)
    firstname = StringField('givenName')
    lastname = StringField('sn')
    common_names = ListField('cn')

    class Meta:
        objectclasses = ['top', 'inetOrgPerson']
        dn_format = 'uid=%(uid)s,ou=people'
        primary_dnprefix = 'ou=people'
        secondary_dnprefix = 'ou=people'
        identifying_attrs = ['uid']
        searchable_fields = ['uid', 'givenName', 'sn']
```

If your LDAP is anonymous, you can start using this model right away:

```
>>> for person in Person.list():
...     print(person.uid)
...
liam
derek
john
```

Otherwise, set an active connection first.

```
Connection.set_connection(Connection.connect())
```

The connection will be set statically for all Models to use.

1.3 Defining Models and Fields

1.3.1 Models

You will define all of your different kinds of objects using `LDAPNode`. An `LDAPNode` models LDAP objects that must be composed of at least two `objectClasses`.

Let us imagine that we are trying to represent the people objects in our directory. We will define a `Person` class. Our `Person` will contain a `Field` for each attribute that a `Person` has.

```
from ldapper import fields
from ldapper.ldapnode import LDAPNode

class BaseModel(LDAPNode):
```

(continues on next page)

(continued from previous page)

```

connection = Connection

class Person(BaseModel):
    uid = fields.StringField('uid', primary=True)
    uidnumber = fields.IntegerField('uidNumber')
    firstname = fields.StringField('givenName')
    lastname = fields.StringField('sn')
    email_addresses = fields.ListField('mailLocalAddress')
    photo = fields.BinaryField('jpegPhoto', optional=True)

    class Meta:
        objectclasses = ['top', 'inetOrgPerson', 'inetLocalMailRecipient']
        dn_format = 'uid=%(uid)s,ou=people'
        primary_dnprefix = 'ou=people'
        secondary_dnprefix = 'ou=people'
        identifying_attrs = ['uid']
        searchable_fields = [
            'uid', 'uidNumber', 'givenName', 'sn', 'mailLocalAddress']

```

Model definition uses the declarative syntax seen in other popular ORMs like SQLAlchemy, Django, or Peewee.

Note: The `Person` gets its connection to your LDAP through the `connection` defined on the `BaseModel`.

1.3.2 Fields

Notice that there are many different field types available to you. All fields are subclasses of `ldapper.fields.Field` and know how to serialize into and out of LDAP.

All fields accept one mandatory argument: the name of the attribute in LDAP.

Fields are required by default. You can pass `optional=True` to the field constructor to make it optional, as is the case for the `photo` attribute.

In the next section we will use our newly-created `Person`.

1.4 The CRUD Operations

This section will make use of the `Person` class we defined in the previous section.

1.4.1 Create

```

person = Person(
    uid='cnorris',
    uidnumber=1337,
    firstname='Chuck',
    lastname='Norris',
    email_addresses=[
        'chuck@example.com',
        'cnorris@megacorp.com',
    ],
)

```

We've now created a person, but not yet saved it. We can check if the object `exists()` or has been saved:

```
>>> person.exists()
False
```

This will query the ldap and look to see if there is something with the DN of this object present.

Let's go and save our person.

```
>>> person.save()
>>> person.exists()
True
```

1.4.2 Read

Tomorrow we come back and want to look up Chuck Norris's LDAP entry. First we will need to fetch him.

```
>>> chuck = Person.fetch('cnorris')
>>> print(chuck)
DN: uid=cnorris,ou=people,dc=umiacs,dc=umd,dc=edu
    uid: cnorris
    firstname: Chuck
    lastname: Norris
    uidnumber: 1337
email_addresses: chuck@example.com
email_addresses: cnorris@megacorp.com
```

If we wanted to get all of our people, we can use `list()`.

```
>>> people = Person.list()
>>> people
[uid=liam,ou=people,dc=umiacs,dc=umd,dc=edu,
 uid=cnorris,ou=people,dc=umiacs,dc=umd,dc=edu]
```

Notice that the `__repr__` is set to use the DN of the object.

1.4.3 Update

In order to make updates to the LDAP, we will need to authenticate.

```
>>> conn = Connection.connect()
Enter a LDAP loginDN or username: liam
Password for LDAP (liam):
>>> Connection.set_connection(conn)
```

Note: An object will use the connection object that was set at the time that it was instantiated. Subsequent changes to the set, static connection will not affect the connection being used by existing objects.

```
>>> chuck = Person.fetch('cnorris')
>>> chuck.firstname = 'Carlos'
```

The careful and the paranoid can see what has changed. LDAP modifications only send modification requests for the attributes that have changed.

```
>>> chuck.diff()
{'firstname': ('Chuck', 'Carlos')}
```

Let's save() our changes.

```
>>> chuck.save()
```

save() calls the Person's validate() method before doing anything else. LDAPNode has a default implementation that just returns True. We can override validate() and get fancy with what it means for an object to be valid.

1.4.4 Destroy

All of that brings us to the final operation: destruction.

We can destroy our person by calling delete().

```
>>> chuck.delete()
>>> chuck.exists()
False
```

1.5 Hooks/Callbacks

There are four callbacks that can be overridden to perform an action before or after an object is added or deleted.

```
class Person(LDAPNode):
    ...

    def _before_add_callback(self):
        self.logger.info('Being called before add.')

    def _after_add_callback(self):
        self.logger.info('Being called after add.')

    def _before_delete_callback(self):
        self.logger.info('Being called before delete.')

    def _after_delete_callback(self):
        self.logger.info('Being called after delete.')
```

1.5.1 _before_add_callback

Gets called just before the LDAPNode is added to the LDAP.

1.5.2 _after_add_callback

Gets called just after the LDAPNode is added to the LDAP.

Note: _before_add_callback() and _after_add_callback() are only called when saving a new object, not when saving existing objects.

1.5.3 `_before_delete_callback`

Gets called just before the LDAPNode is deleted from the LDAP.

1.5.4 `_after_delete_callback`

Gets called just after the LDAPNode is deleted from the LDAP.

1.6 Advanced Topics

1.6.1 Complex Queries

ldapper has rich support for building arbitrarily complex filter queries. Similar to how Django does this, there is a `Q` class that is the building block of queries. `Q` objects can be strung together to build any boolean query imaginable.

The simplest query we could build would be a single condition:

```
from ldapper.query import Q

Person.filter(
    Q(employeetype='Director')
)
```

This will return all the `Person` objects where `employeetype` is equal to `director`. The cool thing here is that when the `Q` object was compiled down, it figured out what the `ldap` field names on `Person` were to build the filter.

We can see that here:

```
>>> Q(employeetype='Director').compile(Person)
'(employeeType=Director)'
```

Let's do something more interesting. Let's return all of the people who are directors with either the first name "Bob" or "Mary".

We would query for those people like this:

```
Person.filter(
    Q(employeetype='Director') & (Q(firstname='Bob') | Q(firstname='Mary'))
)
```

Note: Normal `operator precedence` rules in Python apply concerning `&`, `|`, and parentheses.

`Q` objects can also contain multiple conditions. They will all have to match.

```
Person.filter(
    Q(firstname='Bob', lastname='Smith')
)
```

And of course if you turn logging up to `DEBUG` levels, you can inspect the actual filters that are being generated to return results.

1.7 API Documentation

This documentation is generated directly from the source code.

1.7.1 ldapper.connection

1.7.2 ldapper.exceptions

exception `ldapper.exceptions.LdapperError`

Bases: `Exception`

Base class for exceptions in this module.

A msg MUST be provided.

exception `ldapper.exceptions.AddDNFailed(dn)`

Bases: `ldapper.exceptions.LdapperError`

Exception raised when we failed to add a DN to the LDAP

dn -- DN that failed to be added

msg -- explanation of the error

exception `ldapper.exceptions.ArgumentError(msg)`

Bases: `ldapper.exceptions.LdapperError`

Exception raised when the arguments to a function are invalid

msg -- explanation of the error

exception `ldapper.exceptions.DuplicateValue(attr, value)`

Bases: `ldapper.exceptions.LdapperError`

Tried to write a duplicate value to the LDAP

attr -- Attribute name

value -- Attribute value

exception `ldapper.exceptions.NoSuchAttrValue(dn, attribute, value)`

Bases: `ldapper.exceptions.LdapperError`

Exception raised when a DN does not have a given value.

dn -- the DN that did not exist

attribute -- the attribute that is not present

value -- the value that the attribute was expected to have

msg -- explanation of the error

exception `ldapper.exceptions.NoSuchDN(dn)`

Bases: `ldapper.exceptions.LdapperError`

Exception raised when a DN does not exist.

dn -- the DN that did not exist

msg -- explanation of the error

exception `ldapper.exceptions.InvalidDN(obj, name, dn)`

Bases: `ldapper.exceptions.LdapperError`

Exception raised when a DN does not meet RFC 4514 syntax

dn -- the invalid dn

msg -- explanation of the error

1.7.3 ldapper.fields

1.7.4 ldapper.ldapnode

1.7.5 ldapper.query

class `ldapper.query.Q(**conditions)`

Bases: `object`

Query class used to build arbitrarily complex query filters.

Q objects are strung together and then compiled once we are told what the concrete LDAPNode class is going to be.

The key to the Q class is that it appends conditions when of the same type without creating another level of hierarchy, and it spawns child levels only when the operations (And, Or) switches.

compile (*cls*)

check_type_compat (*other*)

class `ldapper.query.And(ops)`

Bases: `ldapper.query.Q`

compile (*cls*)

class `ldapper.query.Or(ops)`

Bases: `ldapper.query.Q`

compile (*cls*)

1.7.6 ldapper.utils

CHAPTER 2

Indices and tables

- `genindex`
- `modindex`
- `search`

I

`ldapper.exceptions`, [9](#)
`ldapper.query`, [10](#)

A

AddDNFailed, 9
And (*class in ldapper.query*), 10
ArgumentError, 9

C

check_type_compat() (*ldapper.query.Q method*),
10
compile() (*ldapper.query.And method*), 10
compile() (*ldapper.query.Or method*), 10
compile() (*ldapper.query.Q method*), 10

D

DuplicateValue, 9

I

InvalidDN, 9

L

ldapper.exceptions (*module*), 9
ldapper.query (*module*), 10
LdapperError, 9

N

NoSuchAttrValue, 9
NoSuchDN, 9

O

Or (*class in ldapper.query*), 10

Q

Q (*class in ldapper.query*), 10