laurelin Documentation

Release 2.0.4

Alex Shafer

Apr 16, 2020

Contents

1 User Docs **3**
- 1.1 Features Overview . . . 3
- 1.2 Missing/incomplete features . . . 4
- 1.3 Walkthrough . . . 4
  - 1.3.1 Navigating . . . 4
  - 1.3.2 Getting Started . . . 5
  - 1.3.3 LDAP Methods Intro . . . 5
  - 1.3.4 LDAPObject Methods Intro . . . 6
- 1.4 Relative Searching . . . 7
- 1.5 Attributes Dictionaries . . . 7
- 1.6 Modify Operations . . . 8
  - 1.6.1 Raw modify methods . . . 8
  - 1.6.2 Strict modification and higher-level modify functions . . . 8
- 1.7 Global Defaults, LDAP instance attributes, and LDAP constructor arguments . . . 9
- 1.8 Basic usage examples . . . 10
  - 1.8.1 1. Connect to local LDAP instance and iterate all objects . . . 10

2 Simple Search Filters **11**

3 Extensions **13**
- 3.1 Laurelin Extensions . . . 13
- 3.2 LDAP Extensions . . . 13
- 3.3 LDAPObject Extensions . . . 13

4 Config Files **15**
- 4.1 Intro . . . 15
- 4.2 Global Section . . . 16
- 4.3 Extensions Section . . . 16
- 4.4 Connection Section . . . 16
- 4.5 Objects Section . . . 16
- 4.6 Global vs. Connection . . . 17
- 4.7 Load Order . . . 17
- 4.8 Using Dicts Directly . . . 17

5 Creating Extensions **19**
- 5.1 Extension System . . . 20
  - 5.1.1 Extension Classes . . . 20

5.1.2 Schema and Controls Classes . . . 20
5.1.3 Depending on Extensions . . . 21
5.1.4 Publishing Extensions . . . 21
5.2 LDAP Extensions . . . 22
5.3 Controls . . . 23
5.4 Schema . . . 23
5.4.1 Object Classes and Attribute Types . . . 24
5.4.2 Matching Rules . . . 24
5.4.3 Syntax Rules . . . 25
5.4.4 Schema/Controls Registration System . . . 26
5.5 OIDs . . . 26
5.6 Validators . . . 26
5.6.1 SchemaValidator . . . 26
5.7 Class Diagram . . . 27

6 Controls **29**
6.1 Using Controls . . . 29
6.2 Defining Controls . . . 30

7 Changelog **31**
7.1 2.0.4 . . . 31
7.2 2.0.3 . . . 31
7.3 2.0.2 . . . 31
7.4 2.0.1 . . . 31
7.5 2.0.0 . . . 32
7.6 1.5.3 . . . 33
7.7 1.5.2 . . . 33
7.8 1.5.0 . . . 33
7.9 1.4.1 . . . 33
7.10 1.4.0 . . . 33
7.11 1.3.1 . . . 34
7.12 1.3.0 . . . 34
7.13 1.2.0 . . . 34
7.14 1.1.0 . . . 34

8 Reference **35**
8.1 laurelin package . . . 35
8.1.1 Subpackages . . . 35
8.1.1.1 laurelin.extensions package . . . 35
8.1.1.2 laurelin.ldap package . . . 35
8.1.2 Module contents . . . 37

9 Laurelin OID Space **39**
9.1 Namespaces . . . 39
9.2 Objects . . . 39

10 Testing Setup **41**
10.1 System . . . 41
10.2 SASL . . . 41
10.2.1 SASL config ldif . . . 41
10.2.2 Adding sasl user password with . . . 42
10.2.3 SASL auth control test case . . . 42
10.3 LDAPS/StartTLS . . . 42

11 Indices and tables **43**

Python Module Index **45**

Index **47**

Laurelin is a pure-Python ORM-esque LDAP client. Check out the *user docs* to get started. View the source on GitHub.

CHAPTER 1

User Docs

- *Features Overview*
- *Missing/incomplete features*
- *Walkthrough*
  - *Navigating*
  - *Getting Started*
  - *LDAP Methods Intro*
  - *LDAPObject Methods Intro*
- *Relative Searching*
- *Attributes Dictionaries*
- *Modify Operations*
  - *Raw modify methods*
  - *Strict modification and higher-level modify functions*
- *Global Defaults, LDAP instance attributes, and LDAP constructor arguments*
- *Basic usage examples*
  - *1. Connect to local LDAP instance and iterate all objects*

1.1 Features Overview

- Fully compliant with RFC 4510 and its children.
- Pure Python codebase, meaning that it can be used with Python implementations other than CPython.

- Tested against CPython 2.7, 3.3, 3.4, 3.5, 3.6, PyPy, and PyPy3.
- Simplified filter syntax (optional, standard filter syntax is fully supported and used by default)
- Pythonic attributes input and presentation. It's just a dictionary.
- Exceedingly easy relative searching. All objects have a suite of search methods which will automatically pass the object's DN as the search base. In many cases, you wont have to pass *any* arguments to search methods.
- Similarly, all objects have a suite of modify methods which allow you to change attributes on already-queried objects without having to pass their DN again.
- Intelligent modification will never send existing attribute values to the server, nor will it request deletion of attribute values that do not exist. This prevents many unnecessary server errors. Laurelin will go as far as to query the object for you before modifying it to ensure you don't see pointless errors (if you want it to).
- Custom validation. You can define validators which check new objects and modify operations for correctness before sending them to the server. Since you control this code, this can be anything from a simple regex check against a particular attribute value, to a complex approval queue mechanism.
- Highly extensible. New methods can easily and safely be bound to base classes.
- Seamless integration of controls. Once defined, these are just new keyword arguments on particular methods, and additional attributes on the response object.
- Includes Python implementations of standard schema elements. This conveys many benefits:
  - Allows changes to be validated *before* sending the server
  - Allows matching rules to be used to compare attribute values locally. Many attribute types are case-insensitive and have other rules meaning that the standard Python `==` or `in` operators won't tell you what you want to know. Laurelin makes them work according to these rules.

1.2 Missing/incomplete features

Some lesser-used features of the LDAP protocol have not yet been implemented or are incomplete. Check the GitHub issues to see if your use case is affected. Please add a comment if so, or open a new issue if you spot anything else. PRs are always welcome.

1.3 Walkthrough

Note: I'm assuming that if you're here, you're already pretty familiar with LDAP fundamentals. If you don't know how to write a search filter, you may want to do some more reading on LDAP before continuing.

1.3.1 Navigating

Just about everything you need for routine user tasks is available in the `laurelin.ldap` package. *`laurelin.ldap.exceptions`* contains all exception definitions which you may need to import to catch, but even some common ones are included in `laurelin.ldap`. Beyond that, you should not need to get into the sub-modules unless you are defining controls, extensions, schema, or validators.

Built-in extensions are stored in the *`laurelin.extensions`* package.

1.3.2 Getting Started

The first thing you should typically do after importing is configure logging and/or warnings. There is a lot of useful information available at all log levels:

```
from laurelin.ldap import LDAP

LDAP.enable_logging()
# Enables all log output on stderr
# It also accepts an optional log level argument, e.g. LDAP.enable_logging(logging.
↪ERROR)
# The function also returns the handler it creates for optional further manual␣
↪handling

import logging

logger = logging.getLogger('laurelin.ldap')
# Manually configure the logger and handlers here using the standard logging module
# Submodules use the logger matching their name, below laurelin.ldap

LDAP.log_warnings()
# emit all LDAP warnings as WARN-level log messages on the laurelin.ldap logger
# all other warnings will take the default action

LDAP.disable_warnings()
# do not emit any LDAP warnings
# all other warnings will take the default action
```

You can then initialize a connection to an LDAP server. Pass a URI string to the `LDAP` constructor:

```
with LDAP('ldap://dir.example.org:389') as ldap:
    # do stuff...

# Its also possible, but not reccommended, to not use the context manager:
ldap = LDAP('ldap://dir.example.org:389')
```

This will open a connection and query the server to find the "base DN" or DN suffix. An empty `LDAPObject` will be created with the base DN and stored as the `base` attribute on the `LDAP` instance. More on this later. For now we will briefly cover the basic LDAP interface which may seem somewhat familiar if you have used the standard python-ldap client before.

1.3.3 LDAP Methods Intro

`LDAP.search()` sends a search request and returns an iterable over instances of `LDAPObject`. Basic arguments are described here (listed in order):

- `base_dn` - the absolute DN to start the search from
- `scope` - One of:
  - `Scope.BASE` - only search `base_dn` itself
  - `Scope.ONE` - search `base_dn` and its immediate children
  - `Scope.SUB` - search `base_dn` and all of its descendents (default)
- `filter` - standard LDAP filter string
- `attrs` - a list of attributes to return for each object

Use `LDAP.get()` if you just need to get a single object by its DN. Also accepts an optional list of attributes.

`LDAP.add()` adds a new object, and returns the corresponding `LDAPObject`, just pass the full, absolute DN and an *attributes dict*

`LDAP.delete()` deletes an entire object. Just pass the full, absolute DN of the object to delete.

The following methods are preferred for modification, however raw *modify methods* are also provided.

All accept the absolute DN of the object to modify, and an *attributes dictionary*.

`LDAP.add_attrs()` adds new attributes.

`LDAP.delete_attrs()` deletes attribute values. Pass an empty values list in the attributes dictionary to delete all values for an attribute.

`LDAP.replace_attrs()` replaces all values for the given attributes with the values passed in the attributes dictionary. Atrributes that are not mentioned are not touched. Passing an empty list removes all values.

For `LDAP.delete_attrs()` and `LDAP.replace_attrs()` you can specify the constant `LDAP.DELETE_ALL` in place of an empty attribute value list to remove all values for the attribute. For example:

```
ldap.replace_attrs('cn=foo,dc=example,dc=org', {'someAttribute': LDAP.DELETE_ALL})
```

If you wish to require the use of the constant instead of an empty list, pass `error_empty_list=True` to the `LDAP` constructor. You can also pass `ignore_empty_list=True` to silently prevent these from being sent to the server (this will be the default behavior in a future release).

1.3.4 LDAPObject Methods Intro

Great, right? But specifying absolute DNs all the time is no fun. Enter `LDAPObject`, and keep in mind the `base` attribute mentioned earlier.

`LDAPObject` inherits from `AttrsDict` to present attributes. This interface is documented *here*.

`LDAPObject` defines methods corresponding to all of the `LDAP` methods, but pass the object's `dn` automatically, or only require the RDN prefix, with the object's `dn` automatically appended to obtain the absolute DN.

`LDAPObject.search()` accepts all the same arguments as `LDAP.search()` except `base_dn` and `scope`. The object's own DN is always used for `base_dn`, and the `relative_search_scope` is always used as the `scope`.

`LDAPObject.find()` is more or less a better `LDAPObject.get_child()`. It looks at the object's `relative_search_scope` property to determine the most efficient way to find a single object below this one. It will either do a *BASE* search if `relative_seach_scope=Scope.ONE` or a *SUBTREE* search if `relative_search_Scope=Scope.SUB`. It is an error to use this method if `relative_search_scope=Scope.BASE`.

`LDAPObject.get_child()` is analagous to `LDAP.get()` but it only needs the RDN, appending the object's own DN as mentioned earlier. (Note that `LDAPObject.get()` inherits from the native `dict.get()`)

`LDAPObject.add_child()` is analagous to `LDAP.add()` again accepting an RDN in place of a full absolute DN.

Use `LDAPObject.get_attr()` like `dict.get()` except an empty list will always be returned as default if the attribute is not defined.

`LDAPObject`'s modify methods update the server first, then update the local attributes dictionary to match if successful. `LDAPObject.add_attrs()`, `LDAPObject.delete_attrs()`, and `LDAPObject.replace_attrs()` require only a new attributes dictionary as an argument, of the same format as for the matching `LDAP` methods.

`LDAPObject` Examples:

```
people = ldap.base.get_child('ou=people')

print(people['objectClass'])
# ['top', 'organizationalUnit']

people.add_attrs({'description':['Contains all users']})

# list all users
for user in people.search(filter='(objectClass=posixAccount)'):
    print(user['uid'][0])
```

1.4 Relative Searching

All objects have `LDAPObject.search()` and `LDAPObject.find()` methods which utilize the `relative_search_scope` attribute of the object. `relative_search_scope` can be passed as a keyword to any method that creates new objects, including `LDAP.obj()`, `LDAP.get()`, `LDAP.search()`, `LDAP.add()`, `LDAPObject.obj()`, `LDAPObject.find()`, `LDAPObject.search()`, `LDAPObject.get_child()`, and `LDAPObject.add_child()`.

When you create an object from another `LDAPObject` and you *don't* specify the `relative_search_scope`, it is automatically inherited from the parent object. When you create an object from an `LDAP` method, it defaults to `Scope.SUB`.

The real win with this feature is when your tree is structured such that you can set this to `Scope.ONE` as this conveys significant performance benefits, especially when using `LDAPObject.find()`. This allows laurelin to to construct the absolute DN of the child object and perform a highly efficient *BASE* search.

1.5 Attributes Dictionaries

This common interface is used both for input and output of LDAP attributes. In short: dict keys are attribute names, and dict values are a `list` of attribute values. For example:

```
{
    'objectClass': ['posixAccount', 'inetOrgPerson'],
    'uid': ['ashafer01'],
    'uidNumber': ['1000'],
    'gidNumber': ['100'],
    'cn': ['Alex Shafer'],
    'homeDirectory': ['/home/ashafer01'],
    'loginShell': ['/bin/zsh'],
    'mail': ['ashafer01@example.org'],
}
```

Note that there is an `AttrsDict` class defined - there is **no requirement** to create instances of this class to pass as arguments, though you are welcome to if you find the additional methods provided this class convenient, such as `AttrsDict.get_attr()`. Further, it overrides `dict` special methods to enforce type requirements and enable case-insensitive keys.

Also note that when passing an attributes dictionary to `LDAP.replace_attrs()` or `LDAP.delete_attrs()` it is legal to specify the constant `LDAP.DELETE_ALL` in place of a value list.

1.6 Modify Operations

1.6.1 Raw modify methods

`LDAP.modify()` and `LDAPObject.modify()` work similarly to the modify functions in python-ldap, which in turn very closely align with how modify operations are described at the protocol level. A list of `Mod` instances is required with 3 arguments:

1. One of the `Mod` constants which describe the operation to perform on an attribute:

- `Mod.ADD` adds new attributes/values
- `Mod.REPLACE` replaces all values for an attribute, creating new attributes if necessary
- `Mod.DELETE` removes attributes/values.

2. The name of the attribute to modify. Each entry may only modify one attribute, but an unlimited number of entries may be specified in a single modify operation.
3. A list of attribute values to use with the modify operation or the constant `LDAP.DELETE_ALL`:

- The list may be empty for `Mod.REPLACE` and `Mod.DELETE`, both of which will cause all values for the given attribute to be removed from the object. The list may not be empty for `Mod.ADD`. You can also specify the constant `LDAP.DELETE_ALL` in place of any empty list. If you wish to warn about empty lists or require the use of the constant, pass `warn_empty_list=True` or `error_empty_list=True` to the `LDAP` constructor. You can also pass `ignore_empty_list=True` to silently prevent these from being sent to the server (this will be the default behavior in a future release).
- A non-empty list for `Mod.ADD` lists all new attribute values to add
- A non-empty list for `Mod.DELETE` lists specific attribute values to remove
- A non-empty list for `Mod.REPLACE` indicates ALL new values for the attribute - all others will be removed.

Example custom modify operation:

```
from laurelin.ldap.modify import Mod

ldap.modify('uid=ashafer01,ou=people,dc=example,dc=org', [
    Mod(Mod.ADD, 'mobile', ['+1 401 555 1234', '+1 403 555 4321']),
    Mod(Mod.ADD, 'homePhone', ['+1 404 555 6789']),
    Mod(Mod.REPLACE, 'homeDirectory', ['/export/home/ashafer01']),
])
```

Using an `LDAPObject` instead:

```
ldap.base.obj('uid=ashafer01,ou=people').modify([
    Mod(Mod.DELETE, 'mobile', ['+1 401 555 1234']),
    Mod(Mod.DELETE, 'homePhone', LDAP.DELETE_ALL), # delete all homePhone values
])
```

Again, an arbitrary number of `Mod` entries may be specified for each `modify` call.

1.6.2 Strict modification and higher-level modify functions

The higher-level modify functions (`add_attrs`, `delete_attrs`, and `replace_attrs`) all rely on the concept of *strict modification* - that is, to only send the modify operation, and to never perform an additional search. By default, strict modification is **disabled**, meaning that, if necessary, an extra search **will** be performed before sending a modify request.

You can enable strict modification by passing `strict_modify=True` to the `LDAP` constructor.

With strict modification disabled, the `LDAP` modify functions will engage a more intelligent modification strategy after performing the extra query: for `LDAP.add_attrs()`, no duplicate values are sent to the server to be added. Likewise for `LDAP.delete_attrs()`, deletion will not be requested for values that are not known to exist. This prevents many unnecessary failures, as ultimately the final semantic state of the object is unchanged with or without such failures. (Note that with `LDAP.replace_attrs()` no such failures are possible)

With the `LDAPObject` modify functions, the situaiton is slightly more complex. Regardless of the `strict_modify` setting, the more intelligent modify strategy will always be used, using at least any already-queried attribute data stored with the object (which could be complete data depending on how the object was originally obtained). If `strict_modify` is disabled, however, another search *may* still be performed to fill in any missing attributes that are mentioned in the passed attributes dict.

The raw `modify` functions on both `LDAP` and `LDAPObject` are unaffected by the `strict_modify` setting - they will always attempt the modify operation exactly as specified.

1.7 Global Defaults, LDAP instance attributes, and LDAP constructor arguments

All of the `LDAP` constructor arguments are set to None by default. In the constructor, any explicitly `is None` arguments are set to their associated global default. These are attributes of the `LDAP` class, have the same name as the argument, upper-cased, and with a `DEFAULT_` prefix (but the prefix wont be repeated).

For example, the `server` argument has global default `LDAP.DEFAULT_SERVER`, and `default_criticality` is `LDAP.DEFAULT_CRITICALITY`.

Most arguments also have an associated instance property. A complete table is below:

Global Default	`LDAP` instance attribute	`LDAP` constructor keyword
`LDAP.DEFAULT_SERVER`	`host_uri`	`server`
`LDAP.DEFAULT_BASE_DN`	`base_dn`	`base_dn`
`LDAP.DEFAULT_FILTER`	none	none
`LDAP.DEFAULT_DEREF_ALIASES`	`default_deref_aliases`	`deref_aliases`
`LDAP.DEFAULT_SEARCH_TIMEOUT`	`default_search_timeout`	`search_timeout`
`LDAP.DEFAULT_CONNECT_TIMEOUT`	`sock_params[0]`	`connect_timeout`
`LDAP.DEFAULT_STRICT_MODIFY`	`strict_modify`	`strict_modify`
`LDAP.DEFAULT_REUSE_CONNECTION`	none	`reuse_connection`
`LDAP.DEFAULT_SSL_VERIFY`	`ssl_verify`	`ssl_verify`
`LDAP.DEFAULT_SSL_CA_FILE`	`ssl_ca_file`	`ssl_ca_file`
`LDAP.DEFAULT_SSL_CA_PATH`	`ssl_ca_path`	`ssl_ca_path`
`LDAP.DEFAULT_SSL_CA_DATA`	`ssl_ca_data`	`ssl_ca_data`
`LDAP.DEFAULT_FETCH_RESULT_REFS`	`default_fetch_result_refs`	`fetch_result_refs`
`LDAP.DEFAULT_FOLLOW_REFERRALS`	`default_follow_referrals`	`follow_referrals`
`LDAP.DEFAULT_SASL_MECH`	`default_sasl_mech`	`default_sasl_mech`
`LDAP.DEFAULT_SASL_FATAL_DOWNGRADE_CHECK`	`default_sasl_fatal_downgrade_check`	`sasl_fatal_downgrade_check`
`LDAP.DEFAULT_CRITICALITY`	`default_criticality`	`default_criticality`
`LDAP.DEFAULT_VALIDATORS`	`validators`	`validators`
`LDAP.DEFAULT_WARN_EMPTY_LIST`	`warn_empty_list`	`warn_empty_list`
`LDAP.DEFAULT_ERROR_EMPTY_LIST`	`error_empty_list`	`error_empty_list`
`LDAP.DEFAULT_IGNORE_EMPTY_LIST`	`ignore_empty_list`	`ignore_empty_list`
`LDAP.DEFAULT_FILTER_SYNTAX`	`default_filter_syntax`	`filter_syntax`
`LDAP.DEFAULT_BUILT_IN_EXTENSIONS_ONLY`	`built_in_extensions_only`	`built_in_extensions_only`

The `LDAP` instance attributes beginning with `default_` are used as the defaults for corresponding arguments on other methods. `default_sasl_mech` is used with `LDAP.sasl_bind()`, `default_criticality` is the default criticality of all controls, the other `default_` attributes are used with `LDAP.search()`.

The `ssl_` prefixed instances attributes are used as the defaults for `LDAP.start_tls()`, as well as the socket configuration when connecting to an `ldaps://` socket.

1.8 Basic usage examples

1.8.1 1. Connect to local LDAP instance and iterate all objects

```
from laurelin.ldap import LDAP

with LDAP('ldapi:///') as ldap:
    ldap.sasl_bind()
    for obj in ldap.base.search():
        print(obj.format_ldif())
```

`LDAP.sasl_bind()` defaults to the `EXTERNAL` mechanism when an `ldapi:` URI is given, which uses the current user for authorization via the unix socket (Known as "autobind" with 389 Directory Server)

CHAPTER 2

Simple Search Filters

Laurelin provides an alternate syntax for search filters that is much simpler than the standard, RFC 4515-compliant, filter syntax. In short, it is a hybrid between SQL logic expressions and standard LDAP filter comparisons.

In the simplest case of a single comparison, the two syntaxes are identical:

Standard	Simple
`(gidNumber=100)`	`(gidNumber=100)`

But when it comes to expressing logic, the Laurelin simplified filter differs quite a bit:

Standard	Simple
`(&(gidNumber<=1000)(!(memberUid=*)))`	`(gidNumber<=1000) AND NOT (memberUid=*)`

Feel free to include parentheses in your simple filters if it helps clarify the logic:

Simple (without extra parens)	Simple (equivalent with extra parens)
`(gidNumber<=1000) AND NOT (memberUid=*)`	`(gidNumber<=1000) AND (NOT (memberUid=*))`

Some more equivalent standard and simple filters:

Standard	Simple
`(&(abc=foo)(|(def=bar)(ghi=jkl)))`	`(abc=foo) AND ((def=bar) OR (ghi=jkl))`
`(|(abc=foo)(&(def=bar)(ghi=jkl)))`	`(abc=foo) OR (def=bar) AND (ghi=jkl)`
`(&(abc=foo)(|(def=bar)(ghi=jkl))(xyz=abc))`	`(abc=foo) AND ((def=bar) OR (ghi=jkl)) AND (xyz=abc)`

By default, Laurelin will interpret your filters with the **unified** filter syntax, meaning you can embed a full RFC 4515-compliant filter anywhere you see a simple comparison in the above examples. This includes as the only element in the filter, making this fully backwards compatible with RFC 4515 standard filters.

Currently available syntaxes are `FilterSyntax.STANDARD` to limit to RFC 4515, `FilterSyntax.SIMPLE` to limit to only simple comparisons within SQL-style logic, and the default `FilterSyntax.UNIFIED`.

If you wish to restrict the syntax, you can do one of the following:

1. Pass `filter_syntax=` to `LDAP.search()` or any other `search` method:

```
from laurelin.ldap import LDAP, FilterSyntax

with LDAP() as ldap:
    search = ldap.search('o=foo', filter='(abc=foo) AND (def=bar)', filter_
↪syntax=FilterSyntax.SIMPLE)
```

2. Pass `filter_syntax=` to the `LDAP` constructor:

```
from laurelin.ldap import LDAP, FilterSyntax

with LDAP(filter_syntax=FilterSyntax.SIMPLE) as ldap:
    search1 = ldap.search('o=foo', filter='(abc=foo) AND (def=bar)')
    search2 = ldap.search('o=bar', filter='(xyz=foo) OR (abc=bar)')
```

3. Set the global default `LDAP.DEFAULT_FILTER_SYNTAX` before instantiating any `LDAP` instances:

```
from laurelin.ldap import LDAP, FilterSyntax

LDAP.DEFAULT_FILTER_SYNTAX = FilterSyntax.STANDARD

with LDAP() as ldap:
    search = ldap.search('o=foo', filter='(&(abc=foo)(def=bar))')

with LDAP('ldap://localhost:10389') as ldap:
    search = ldap.search('o=bar', filter='(|(xyz=foo)(abc=bar))')
```

4. Do either of the two above using *Config Files*.

Note: How is this possible?

Doesn't the filter get sent to the server and parsed there like SQL? No! In LDAP, it is up to the client to parse the filter string into a set of objects that are encoded and sent to the server. If you've got any other ideas for alternate filter syntaxes, please submit a PR!

CHAPTER 3

Extensions

The following class documents show names of available extensions on different instances.

3.1 Laurelin Extensions

Every defined extension has a property in this class. An instance is accessible at `laurelin.ldap.extensions`. For example, to require the base schema:

```
from laurelin.ldap import extensions

extensions.base_schema.require()
```

3.2 LDAP Extensions

These properties are available on `LDAP` instances.

3.3 LDAPObject Extensions

These properties are available on `LDAPObject` instances.

CHAPTER 4

Config Files

- *Intro*
- *Global Section*
- *Extensions Section*
- *Connection Section*
- *Objects Section*
- *Global vs. Connection*
- *Load Order*
- *Using Dicts Directly*

4.1 Intro

Laurelin config files may be YAML or JSON formatted out of the box. You can also supply your own custom decoding function to handle arbitrary formats. The important part is that the file contents decode to a dictionary. Below is an example YAML file:

```
global:
  SSL_CA_PATH: /etc/ldap/cacerts
  IGNORE_EMPTY_LIST: true
extensions:
  - laurelin.extensions.descattrs
  - laurelin.extensions.netgroups
connection:
  server: ldap://dir01.example.org
  start_tls: true
  simple_bind:
```

(continues on next page)

(continued from previous page)

```
    username: testuser
    passowrd: testpassword
  connect_timeout: 30
objects:
  - rdn: ou=people
    tag: posix_user_base
  - rdn: ou=groups
    tag: posix_group_base
  - rdn: ou=netgroups
    tag: netgroup_base
```

You can load and apply such a file by using `config.load_file()`. If a `connection` section was specified, a new connection will be established and returned from the function.

4.2 Global Section

Each key in the global section must correspond to one of the `DEFAULT_` prefixed attributes on `LDAP`. As you can see in the example, the `DEFAULT_` prefix is optional. Not demonstrated by the example is that keys are case-insensitive (that is, they will be upper-cased for you).

4.3 Extensions Section

This is simply a list of extension module names which will get activated when the config file is loaded.

4.4 Connection Section

Keys here are *mostly* corresponding to `LDAP` constructor arguments, however there are a few special ones:

- `start_tls` A boolean option, if set to `true` will execute `LDAP.start_tls()` after opening the connection
- `simple_bind` A dictionary of parameters to pass to `LDAP.simple_bind()`
- `sasl_bind` A dictionary of parameters to pass to `LDAP.sasl_bind()`

Note that `simple_bind` and `sasl_bind` are both optional, and mutually exclude each other. In other words, it is an error to specify both of these keys.

Note that `start_tls` will always occur before any bind (if requested).

4.5 Objects Section

Note: You cannot specify `objects` without also specifying a `connection`

This is a list of dicts where keys correspond to `LDAP.obj()` or `LDAPObject.obj()` arguments. You *must* specify exactly one of `dn` or `rdn`. If `dn` is specified, this will be taken as the full, absolute DN of the object, and parameters will be passed to `LDAP.obj()`. If `rdn` is specified, this will be taken as the RDN relative to the connection's base object, or the base of the tree, and parameters will be passed to `LDAPObject.obj()`.

Also required for all objects is the `tag` key. This is how you will access created objects. For example, to access the first object in the config file example above:

```
ldap = config.load_file('/path/to/file.yaml')
posix_users = ldap.tag('posix_user_base')
```

Its important to note that the server is not queried when creating these objects, so they will not have any local attributes. If you require local attributes, you can all `LDAPObject.refresh()` on the object.

4.6 Global vs. Connection

As mentioned elsewhere in the docs, there is a global config parameter associated with every connection parameter, meaning in a config file you can define your connection parameters in either section. This *does not* have the exact same end functionality, though. In general you should prefer `connection` for the following reasons:

- The connection will not be created when the config file is loaded if you configure everything in `global`
- You cannot define `objects` without defining a `connection`
- You cannot specify `start_tls` or bind parameters globally

However there are cases where it may be desirable to specify everything as a global default. Taking this approach allows you to use the `LDAP` constructor with as few as zero arguments after loading the config. You can still bind as usual by calling `LDAP.simple_bind()` or `LDAP.sasl_bind()` on the connection. You can also manually create objects with `obj()` methods.

4.7 Load Order

Sections are loaded and applied in a specific order:

1. `global`
2. `extensions`
3. `connection`
4. `objects`

You can specify sections in whatever order is convenient in your file. They will *always* be used in the above order.

4.8 Using Dicts Directly

If you already have your configuration parameters in one or more dictionaries, you can apply them directly without going through the file interface. You can pass a dictionary of the same format as in a config file to `config.load_config_dict()`. Like `load_file()`, this will establish and return the new connection if one was defined.

You can also use the other `config` methods to apply dictionary configurations piecemeal. These process fragments of the larger config dictionary. Check the reference docs for details if you need to do this.

CHAPTER 5

Creating Extensions

- *Extension System*
  - *Extension Classes*
  - *Schema and Controls Classes*
  - *Depending on Extensions*
  - *Publishing Extensions*
- *LDAP Extensions*
- *Controls*
- *Schema*
  - *Object Classes and Attribute Types*
  - *Matching Rules*
  - *Syntax Rules*
  - *Schema/Controls Registration System*
- *OIDs*
- *Validators*
  - *SchemaValidator*
- *Class Diagram*

The most important thing to note about "extensions" is that they are not necessarily LDAP extensions. In laurelin, they are simply a module that does any combination of: defining new schema elements, defining new controls, or defining new methods to be attached to `LDAP` or `LDAPObject`.

5.1 Extension System

Extensions live in any importable module or package. They must at minimum define a class called `LaurelinExtension` as follows:

```
from laurelin.ldap import BaseLaurelinExtension

class LaurelinExtension(BaseLaurelinExtension):
    NAME = 'some_name'
```

You'll notice the `BaseLaurelinExtension` here - this is required. It is one of many weapons at your disposal.

5.1.1 Extension Classes

All of these share the same common end-user interface of being exposed as either a property or dynamic attribute on some other instance that the user typically will already use normally. Which class they are attached to depends on the name and base class of the defined extension class. Whether they are accessible as a property (with IDE auto-complete support) or a dynamic attribute depends on how the extension is loaded and defined (more below), but the user API is unchanged either way.

`class LaurelinExtension(BaseLaurelinExtension):` As described above, this is where you define the name of the property or dynamic attribute where all instances of these extension classes can be accessed. One instance of this class is created per Python interpreter when the extesion is first added or used (more on this later) and it is accessible to users at `laurelin.ldap.extensions.<NAME>`.

`class LaurelinLDAPExtension(BaseLaurelinLDAPExtension):` This is where you can bind methods, attributes, etc. that will be attached to `LDAP` by way of property or dynamic attribute with name corresponding to your `LaurelinExtension.NAME`. You can access the parent `LDAP` instance at `self.parent`. Up to one instance is created per `LDAP` instance when the property or dynamic attribute is first accessed on a particular instance.

`class LaurelinLDAPObjectExtension(BaseLaurelinLDAPObjectExtension):` This is where you can bind methods, attributes, etc. that will be attached to `LDAPObject` by way of property or dynamic attribute with name corresponding to your `LaurelinExtension.NAME`. You can access the parent `LDAPObject` instance at `self.parent`. Up to once instance is created per `LDAPObject` instance when the property or dynamic attribute is first accessed on a particular instance.

5.1.2 Schema and Controls Classes

These two simply attempt to register all public attributes defined within them as schema elements or controls. More about actually defining these below, the class signatures should look like this, though:

`class LaurelinSchema(BaseLaurelinSchema):` Define all `SyntaxRule` and `EqualityMatchingRule` classes as local classes within this class. Directly instantiate `ObjectClass` and `AttributeType` with standard spec strings and assign them to class attributes.

`class LaurelinControls(BaseLaurelinControls):` Define all `Control` classes as local classes within this class.

Note that the placement of schema and control definitions is fairly flexible and are not restricted to these 2 classes (but this kind of organization or a variation upon it is suggested). See the Schema and Controls sections below for more details.

Also note that if your schema depends on the base schema, you must require it at the top of your extension like so:

```
from laurelin.ldap import extensions

extensions.base_schema.require()
```

5.1.3 Depending on Extensions

Extension authors may want to duplicate and tailor some or all of this information in their own documentation for users.

There are two ways laurelin can be made aware of extensions:

1. By passing a module name string to `add_extension()`. This will cause the extension class instances to be made available as dynamic attributes.
2. By being defined in `Extensible.AVAILABLE_EXTENSIONS`. A script will automatically generate properties that are inherited by the appropriate parent class (`LDAP` or `LDAPObject`). This has the benefit that IDEs can auto-complete extension instances if the extension is installed (tested with PyCharm). Also defined with your extension is the string module name, so your users do not need to copy this themselves, as well as the pip package name, which will be included in the exception if users attempt to use your extension when its not installed.

There are clear pros and cons to each approach, and extension authors are welcome to instruct users to take either approach. #1 may be preferred during development, or if you do not intend to publish your extension publicly.

One caveat to #2 above if you define schema or controls, is your users will need to explicitly require your extension like so:

```
from laurelin.ldap import extensions

extensions.<NAME>.require()
```

This happens implicitly in the following situations:

- When `add_extension()` is called, as in #1 above
- When the user accesses your `<NAME>` extension property/attribute on `LDAP` or `LDAPObject`, if you defined any extensions to those classes
- Technically happens implicitly when `extensions.<NAME>` is accessed, so if you define any other user-exposed attributes on your `LaurelinExtension` class that all users *must* access, you can instruct them to use that instead.

So if you **require** any of these of your users by way of your own documentation, you can also have them skip the explicit `require()` call.

Regardless of whether your extension is added or defined, your users will need to explicitly add the dependency to their own package. Laurelin will *never* depend on an extension module, and only built-in extensions are guarnateed to be available.

5.1.4 Publishing Extensions

If you are planning on defining any standard LDAP extensions, schema, or controls, I suggest packaging your module under `laurelin.extensions`, which is a namespace package. This allows an exceedingly simple and easy path to eventual merging in as a built-in extension. You are welcome to package under any importable module, though.

If you choose to instruct your users to add your extension, please be sure to write clear and accessible documentation for them.

If you choose to define your extension, please submit a pull request on GitHub. You should include ONLY a ~5 line addition to `Extensible.AVAILABLE_EXTENSIONS`. The dict key should match your `LaurelinExtension.NAME`. The keys in the sub-dictionary should be pretty self-explanatory. Below is a contrived example patch:

```
diff --git a/laurelin/ldap/extensible/base.py b/laurelin/ldap/extensible/base.py
index 593e64b..bd7b233 100644
--- a/laurelin/ldap/extensible/base.py
+++ b/laurelin/ldap/extensible/base.py
@@ -132,6 +132,11 @@ class Extensible(object):
             'pip_package': None,  # built-in
             'docstring': 'Built-in extension defining standard paged results control␣
↪for search'
         },
+        'some_ext': {
+            'module': 'your.extension.module',
+            'pip_package': 'laurelin-some-ext',
+            'docstring': 'A contrived example laurelin extension'
+        },
     }

     ADDITIONAL_EXTENSIONS = {}
```

Please keep your docstrings short. They will be rendered in laurelin's documentation. You may include a Sphinx-formatted shortlink to your own docs.

If you have any questions, problems, or concerns, please open an issue on GitHub.

5.2 LDAP Extensions

When defining an actual LDAP extension with an OID and requiring server support, you'll create the laurelin extension as shown above, but you'll be calling the `LDAP.send_extended_request()` method from your extension methods within your `LaurelinLDAPExtension` or `LaurelinLDAPObjectExtension`.

As you can see, this accepts the OID of the LDAP extension and an optional request value. You can also pass control keywords, and the `require_success` keyword, which will automatically check for success on the final extendedResponse message (and raise an *LDAPError* on failure).

If your LDAP extension expects intermediateResponse messages, you can iterate the return from `LDAP.send_extended_request()`. You can also call `ExtendedResponseHandle.recv_response()` to get only one message at a time (preferred to iteration if you only expect the one extendedResponse message).

The built-in `LDAP.who_am_i()` method is an excellent example of a simple LDAP extension:

```
from laurelin.ldap import LDAP
from laurelin.ldap.protoutils import get_string_component

def who_am_i(self):
    handle = self.send_extended_request(LDAP.OID_WHOAMI, require_success=True,␣
↪**ctrl_kwds)
    xr, res_ctrls = handle.recv_response()
    return get_string_component(xr, 'responseValue')
```

As a laurelin extension this might look like:

```
from laurelin.ldap import BaseLaurelinLDAPExtension
```

(continues on next page)

(continued from previous page)

```
# ...

class LaurelinLDAPExtension(BaseLaurelinLDAPExtension):
    def who_am_i(self):
        handle = self.parent.send_extended_request(...)
        # ...
```

Note the use of `self.parent` to access `LDAP.send_extended_request()`.

5.3 Controls

Extensions may wish to define controls for use on existing methods. You will need to define one or more `Control` classes, see *Defining Controls* for more information about this. The important part for the purposes of this document is where to place those class definitions in your extension module.

You must define a subclass of `LaurelinTransiter`, or the more semantically appropriate but functionally identical `BaseLaurelinControls`. Your subclass must then have local `Control` subclasses defined within it. For example:

```
from laurelin.ldap import BaseLaurelinExtension, BaseLaurelinControls, Control

class LaurelinExtension(BaseLaurelinExtension):
    NAME = 'your_name'

class LaurelinControls(BaseLaurelinControls):
    class YourControl(Control):
        method = ('search',)
        keyword = 'some_kwd'
        REQUEST_OID = '1.2.3.4'
```

Note that controls may alternatively be defined directly in your `LaurelinExtension` class.

5.4 Schema

Extensions may be associated with a set of new schema elements, including object classes, attribute types, matching rules, and syntax rules. Once defined, these will get used automatically by other parts of laurelin, including the `SchemaValidator`, and for comparing items in attribute value lists within an `LDAPObject`.

Like controls, all extension schema elements must be defined as attributes on a subclass of `LaurelinTransiter`. The more semantically appropriate `BaseLaurelinSchema` is provided as well. You can use these base classes to organize your schema and controls however appropriate. Alternatively, you may also define schema elements directly in your `LaurelinExtension` class.

If your schema depends on the laurelin built-in base schema, you must explicitly call `laurelin.ldap.extensions.base_schema.require()` near the top of your extension module.

Below is a simple example of defining a new object class depending on the base schema:

```
from laurelin.ldap import BaseLaurelinExtension, BaseLaurelinControls, ObjectClass,␣
↪extensions

extensions.base_schema.require()
```

(continues on next page)

(continued from previous page)

```
class LaurelinExtension(BaseLaurelinExtension):
    NAME = 'your_name'

class LaurelinSchema(BaseLaurelinSchema):
    MY_COMPANY_USER = ObjectClass('''
    ( 1.2.3.4 NAME 'myCompanyUser' SUP inetOrgPerson STRUCTURAL
      MUST ( companyAttribute $ anotherAttribute )
      MAY description
    ''')
```

The superclass of `inetOrgPerson` makes this example require the base schema. All schema instance elements must be defined as class attributes in this manner (for object classes and attribute types), and all class elements must be defined below the `LaurelinSchema` class (for syntax rules and matching rules).

5.4.1 Object Classes and Attribute Types

Creating object classes and attribute types is very simple. Just take the standard LDAP specification and pass it to the appropriate class constructor. Examples from the netgroups extension:

```
from laurelin.ldap.objectclass import ObjectClass
from laurelin.ldap.attributetype import AttributeType

ObjectClass('''
( 1.3.6.1.1.1.2.8 NAME 'nisNetgroup' SUP top STRUCTURAL
  MUST cn
  MAY ( nisNetgroupTriple $ memberNisNetgroup $ description ) )
''')

AttributeType('''
( 1.3.6.1.1.1.1.14 NAME 'nisNetgroupTriple'
  DESC 'Netgroup triple'
  EQUALITY caseExactMatch
  SYNTAX 1.3.6.1.1.1.0.0 )
''')
```

5.4.2 Matching Rules

Defining matching rules takes a little more effort. Matching rules must subclass `EqualityMatchingRule`. Required class attributes include:

- `OID` - the numeric OID of this rule (see section below about OIDs).
- `NAME` - the name of the rule. Must also be globally unique. This is usually how matching rules are referenced in attribute type specs (see `caseExactMatch` in above example).
- `SYNTAX` - the numeric OID of the syntax rule that assertion values must match.

Matching rule classes may also optionally define the following attribute:

- `prep_methods` - a sequence of callables that will be used to prepare both the attribute value and assertion value for comparison. These will typically be defined in `laurelin.ldap.rfc4518`. The initial attribute/assertion value will be passed into the first item in the sequence, and the return from each is passed into the next item.

If you prefer, you can also override the `MatchingRule.prepare()` method on your matching rule class.

You may also wish to override `EqualityMatchingRule.do_match()`. This is passed the two prepared values and must return a boolean. Overriding `MatchingRule.match()` *is not recommended*.

Below is an example matching rule from `laurelin.extensions.base_schema`:

```
from laurelin.ldap.rules import EqualityMatchingRule
from laurelin.ldap import rfc4518

 class numericStringMatch(EqualityMatchingRule):
     OID = '2.5.13.8'
     NAME = 'numericStringMatch'
     SYNTAX = '1.3.6.1.4.1.1466.115.121.1.36'
     prep_methods = (
         rfc4518.Transcode,
         rfc4518.Map.characters,
         rfc4518.Normalize,
         rfc4518.Prohibit,
         rfc4518.Insignificant.numeric_string,
     )
```

5.4.3 Syntax Rules

Syntax rules must subclass `SyntaxRule`, although in almost all cases you can use `RegexSyntaxRule`. If you do not use a regular expression, you must override `SyntaxRule.validate()`, which receives a single string argument, and must raise *`InvalidSyntaxError`* when it is incorrect.

In all cases, you must define the following attributes on your syntax rule class:

- `OID` - the numeric OID of the rule (See section below about OIDs).
- `DESC` - a brief description of the rule. This is mainly used in exception messages.

Regex syntax rules must also define:

- `regex` - the regular expression.

Below are examples from `laurelin.extensions.base_schema`:

```
from laurelin.ldap.rules import SyntaxRule, RegexSyntaxRule
from laurelin.ldap.exceptions import InvalidSyntaxError
import six

 class DirectoryString(SyntaxRule):
     OID = '1.3.6.1.4.1.1466.115.121.1.15'
     DESC = 'Directory String'

     def validate(self, s):
         if not isinstance(s, six.string_types) or (len(s) == 0):
             raise InvalidSyntaxError('Not a valid {0}'.format(self.DESC))

 class Integer(RegexSyntaxRule):
     OID = '1.3.6.1.4.1.1466.115.121.1.27'
     DESC = 'INTEGER'
     regex = r'^-?[1-9][0-9]*$'
```

5.4.4 Schema/Controls Registration System

Schema and controls go through an identical 2-step registration system. The `LaurelinTransiter` class first stores a list of all schema and control attributes mapped to the module name that defined them. This occurs when the class is defined, i.e. at import time.

The `LaurelinRegistrar.require()` method then invokes the `.register()` method on each schema element or control class defined in the same module. This causes the element to be mapped according to its class, name, and OID - which are ultimately what is needed for laurelin to make use of the object.

5.5 OIDs

Controls and schema elements all need an OID to be defined. You should obtain a Private Enterprise Number from IANA for any OIDs that you publish to the internet (and probably for any that you don't as well). This is completely free and usually only takes a few days to process.

The OID you get from IANA should be used as the root of your namespace, and you can define the structure below it as you see fit.

5.6 Validators

Validators must subclass `Validator`. The public interface includes `Validator.validate_object()` and `Validator.validate_modify()`. You will usually just want to override these, however they do include a default implementation which checks all attributes using the abstract `Validator._validate_attribute()`. Check method docs for more information about how to define these.

When defining validators in your extension, you can ensure your users don't need to import the module again by attaching the class to your `LaurelinExtension` class like so:

```
from laurelin.ldap import BaseLaurelinExtension, Validator

class LaurelinExtension(BaseLaurelinExtension):
    NAME = 'my_ext'

    class MyValidator(Validator):
        # ...
        pass
```

Users can then access it like so:

```
from laurelin.ldap import LDAP, extensions

with LDAP('ldaps://dir.example.org', validators=[extensions.my_ext.MyValidator]) as␣
↪ldap:
    # do stuff
```

5.6.1 SchemaValidator

Laurelin ships with `SchemaValidator` which, when applied to a connection, automatically checks write operations for schema validity *before* sending the request to the server. This includes any schema you define in your extensions. Users can enable this like so:

```
from laurelin.ldap import LDAP
from laurelin.ldap.schema import SchemaValidator

with LDAP('ldaps://dir.example.org', validators=[SchemaValidator]) as ldap:
    # do stuff
```

5.7 Class Diagram

The extension subsystem has several interconnecting classes. Blue are auto-generated classes, and green are defined in extension modules. Unlabeled arrows indicate class inheritance or are self-explanatory.

CHAPTER 6

Controls

- *Using Controls*
- *Defining Controls*

Many LDAP users may be unfamiliar with controls. RFC4511 defines *controls* as "providing a mechanism whereby the semantics and arguments of existing LDAP operations may be extended." In other words, they can:

1. Instruct the server to process a method differently
2. Add new arguments to methods to control the altered processing
3. Add additional data to the response to a method call

It is important to note that both the server and client must mutually support all controls used. Laurelin will automatically check for server support when using controls.

6.1 Using Controls

Once controls have been *defined*, they are very easy to use. Each control has a `keyword` and optionally a `response_attr`.

The `keyword` can be passed as a keyword argument to specific methods. The value type and format is up to the control implementation. Whatever value the control expects can be wrapped in `critical` or `optional` to declare the criticality of the control.

If defined, the `response_attr` will be set as an attribute on the object returned from the method call.

For search response controls, the control value will be set on the individual `LDAPObject` if it appeared on the associated search result entry. If it appeared on the search results done message, the control value will be set on the iterator object.

In the highly unusual case that a response control is set on a search result reference message, the control values will be inaccessible if `fetch_result_refs` is set to True. A warning will be issued in this case.

If `fetch_result_refs` is set to False, the response control values will be set on the `SearchReferenceHandle` that is yielded from the results iterator.

An *`LDAPSupportError`* will be raised if the control is marked critical and the server does not support it.

6.2 Defining Controls

Controls must subclass Control. As soon as they are defined as a subclass of Control, they are ready to use. Controls must define at least:

- `Control.method`, a tuple of method names that this control supports. Current method names are *bind*, *search*, *compare*, *add*, *delete*, *mod_dn*, *modify*, and *ext* (extended request). Note that these method names do not necessarily correspond directly to `LDAP` method names. Even when they do, other methods may call the base method and pass through control keywords. For example, `LDAPObject.find()` ends up passing any control keywords through into `LDAP.search()` (which matches the *search* method). The *bind* method is used by both `LDAP.simple_bind()` and `LDAP.sasl_bind()`.
- `Control.keyword`, the keyword argument to be used for the request control.
- `Control.REQUEST_OID` the OID of the reuqest control. If the control has criticality, the OID must be listed in the supportedControl attribute of the root DSE of the server at runtime.

If there is an associated response control, also define the following:

- `Control.response_attr`, the name of the attribute which will be set on objects returned from the method.
- `Control.RESPONSE_OID` the OID of the response control. This may be equal to `Control.REQUEST_OID` depending on the spec. This must match the controlType of the response control to be properly set.

Most controls will not need to override methods if only strings are used for request and response values. However, if it is desirable to use a more complex data structure as a control value, you can override the `Control.prepare()` method to accept this structure as its first argument. You will need to process this into a single string for transmission to the server, and pass it into, and return, the base `Control.prepare()`. The second argument is a boolean describing criticality, and must also be passed into the base method.

To return a more complex value for the response, you can override the `Control.handle()` method. This will be passed the response control value string, and the return will be assigned to the `response_attr` attribute on the returned object.

CHAPTER 7

Changelog

7.1 2.0.4

Released 2019.05.30

- Switch to an internal pyasn1
- Fix issue with binary data

7.2 2.0.3

Released 2019.02.14

- No code changes. Clarified stability guarantee for `laurelin.extensions`

7.3 2.0.2

Released 2019.02.12

- Fix: make extension requirements align with specification
- Update documentation with OID information

7.4 2.0.1

Released 2019.02.09

- Fix: Correctly request no attributes be returned for `LDAP.exists()`

7.5 2.0.0

Released 2018.11.17

- Empty lists in a `replace` or `delete` modify operation are now **ignored by default**
  - To delete all attribue values in a replace or delete, use `DELETE_ALL` introduced in version 1.2.0.
  - To restore the previous functionality, you can set the global default `LDAP.DEFAULT_IGNORE_EMPTY_LIST = False`, or restore on a per-connection basis by passing `ignore_empty_list=False` to the `LDAP()` constructor.
  - The rationale for this change is a) improved semantics, and b) eliminates unexpected behavior in cases like applying a filter to determine a list to remove (which may result in an empty list, meaning no items should be removed)
- Extensions API has been changed, both for users and creators of extensions:
  - Rather than attaching new attributes directly to the LDAP or LDAPObject class, a property (or dynamic attribute) is made available on those classes for each extension, which provides access to an object exposing those same attributes.
  - Many extension attributes have been renamed to avoid semantic duplication introduced by this change. For example `ldap.get_netgroup_users()` should be replaced with `ldap.netgroups.get_users()`.
  - The `descattrs` extension has been changed slightly to work better with these new changes. Description attributes can now be accessed and modified like so (no additional imports necessary):

```
o = ldap.base.obj('cn=metadata')
print(o.descattrs['some_attr'])
# ['value1', 'value2']

o.descattrs.add({'some_attr': ['value3']})
print(o.descattrs['some_attr'])
# ['value1', 'value2', 'value3']

# these also work now:

'some_attr' in o.descattrs

for attr in o.descattrs:
```

  - Docs have been updated with information about creating extensions.
  - Internal changes around loading of schema elements and controls
- Base schema changes:
  - The base schema will now be automatically loaded when needed. At present, this includes:
    - When checking for the presence of a value in an attribute list
    - When a `SchemaValidator` is initialized
    - When the `netgroups` extension is used
  - The base schema is no longer defined in `laurelin.ldap.schema`. It now is housed in a built-in extension. If previously using `import laurelin.ldap.schema` or similar to enable client-side schema checking, this should be replaced with something like the following:

```
from laurelin.ldap import extensions
extensions.base_schema.require()
```

However, as stated above, this will not be necessary for almost all use cases.

- Properly documented the public API definition

7.6 1.5.3

Release 2018.08.30

- Add python 3.7 support

7.7 1.5.2

Released 2018.06.15

1.5.1 was built off of the wrong branch and will be removed.

- Minor fix: Added FilterSyntax to all
- Doc update: added dependent info section to readme

7.8 1.5.0

Released 2018.06.09

- Added new simple filter syntax
- Switched default filter syntax to UNIFIED which should be backwards compatible with standard RFC 4515 filters

Special thanks to @jpypi for authoring the new grammar

7.9 1.4.1

Released 2018.05.31

- Fix: Checked for failed import of AF_UNIX to improve Windows support
- Fix: Required latest pure-sasl

7.10 1.4.0

Released 2018.05.29

- Validation updates:
  - Added `LDAP.disable_validation()` which creates a context with any or all validators skipped
  - Added an `ldap_conn` attribute to validator instances to allow validators to query the server

  - Allowed passing a class as well as an instance with the `validators` constructor keyword
- Greatly improved handling of unsolcitied messages (message ID 0)
- Fix: enforce maximum length for attribute types
- Fix: SASL auth issues with pure-sasl 0.5.1+

7.11 1.3.1

Released 2018.04.01

- Fixed logic bug in `SchemaValidator` when an object has two or more object classes that require one or more of the same attributes
- Fixed: allowed string `some.module.Class` specification for validators in config files

7.12 1.3.0

Released 2018.03.22

- Added config file support, see `laurelin.ldap.config`
- Fixed: ensured extensions can be safely activated multiple times
- Fixed: `Mod` constants `repr` updated for consistency

7.13 1.2.0

Released 2018.03.16

- Add DELETE_ALL to use as an attribute value list with modify, replace_attrs, and delete_attrs
- Added new constructor keywords to alter the behavior of empty value lists for modify, replace_attrs, and delete_attrs:
  - `ignore_empty_list` to silently ignore empty value lists and not send them to the server. This will be enabled by default in a future release.
  - `error_empty_list` to raise an exception when an empty value list is passed.
  - `warn_empty_list` to emit a warning when an empty value list is passed.

7.14 1.1.0

Released 2018.03.12

Initial stable API.

CHAPTER 8

Reference

8.1 laurelin package

8.1.1 Subpackages

8.1.1.1 laurelin.extensions package

Submodules

laurelin.extensions.base_schema module

laurelin.extensions.descattrs module

laurelin.extensions.netgroups module

laurelin.extensions.pagedresults module

Module contents

8.1.1.2 laurelin.ldap package

Submodules

laurelin.ldap.base module

laurelin.ldap.config module

laurelin.ldap.exceptions module

exception laurelin.ldap.exceptions.**Abandon**
Bases: Exception

Can be raised to cleanly exit a context manager and abandon unread results

exception laurelin.ldap.exceptions.**ConnectionAlreadyBound**
Bases: *laurelin.ldap.exceptions.InvalidBindState*

Only raised by LDAP.*Bind methods if the connection is already bound when called

exception laurelin.ldap.exceptions.**ConnectionUnbound**
Bases: *laurelin.ldap.exceptions.InvalidBindState*

Raised when any server operation is attempted after a connection is unbound/closed

exception laurelin.ldap.exceptions.**InvalidBindState**
Bases: *laurelin.ldap.exceptions.LDAPError*

Base class for exceptions related to bind state

exception laurelin.ldap.exceptions.**InvalidSyntaxError**
Bases: *laurelin.ldap.exceptions.LDAPValidationError*

Raised when syntax validation fails

exception laurelin.ldap.exceptions.**LDAPConnectionError**
Bases: *laurelin.ldap.exceptions.LDAPError*

Error occurred creating connection to the LDAP server

exception laurelin.ldap.exceptions.**LDAPError**
Bases: Exception

Base class for all exceptions raised by laurelin

exception laurelin.ldap.exceptions.**LDAPExtensionError**
Bases: *laurelin.ldap.exceptions.LDAPError*

Error occurred setting up an extension module

exception laurelin.ldap.exceptions.**LDAPSASLError**
Bases: *laurelin.ldap.exceptions.LDAPError*

Error occurred involving the SASL client

exception laurelin.ldap.exceptions.**LDAPSchemaError**
Bases: *laurelin.ldap.exceptions.LDAPError*

Error relating to setting up the LDAP schema

exception laurelin.ldap.exceptions.**LDAPSupportError**
Bases: *laurelin.ldap.exceptions.LDAPError*

A feature is not supported by the server

exception laurelin.ldap.exceptions.**LDAPTransactionError**
Bases: *laurelin.ldap.exceptions.LDAPError*

Raised by actions not included in a modify transaction

exception laurelin.ldap.exceptions.**LDAPUnicodeWarning**
Bases: *laurelin.ldap.exceptions.LDAPWarning*, UnicodeWarning

Warning category for unicode issues relating to LDAP

exception laurelin.ldap.exceptions.**LDAPUnsolicitedMessage**(*lm*, *exc_msg*)
Bases: Exception

Raised when a message with ID 0 is returned from the server

This may indicate an incompatability between laurelin and your server distribution and thus is outside the normal exception inheritance chain.

exception laurelin.ldap.exceptions.**LDAPValidationError**
Bases: *laurelin.ldap.exceptions.LDAPError*

Raised when validation fails

exception laurelin.ldap.exceptions.**LDAPWarning**
Bases: Warning

Generic LDAP warning category

exception laurelin.ldap.exceptions.**MultipleSearchResults**
Bases: *laurelin.ldap.exceptions.UnexpectedSearchResults*

Got multiple search results when exactly one was required

exception laurelin.ldap.exceptions.**NoSearchResults**
Bases: *laurelin.ldap.exceptions.UnexpectedSearchResults*

Got no search results when one or more was required

exception laurelin.ldap.exceptions.**ProhibitedCharacterError**
Bases: *laurelin.ldap.exceptions.LDAPError*

Raised when a prohibited character is detected in RFC4518 string prep

exception laurelin.ldap.exceptions.**TagError**
Bases: *laurelin.ldap.exceptions.LDAPError*

Error with an object tag

exception laurelin.ldap.exceptions.**UnexpectedResponseType**
Bases: *laurelin.ldap.exceptions.LDAPError*

The response did not contain the expected protocol operation

exception laurelin.ldap.exceptions.**UnexpectedSearchResults**
Bases: *laurelin.ldap.exceptions.LDAPError*

Base class for unhandled search result situations

laurelin.ldap.ldapobject module

laurelin.ldap.protoutils module

Module contents

8.1.2 Module contents

CHAPTER 9

Laurelin OID Space

Laurelin IANA Registered Private Enterprise Number:

1.3.6.1.4.1.53450

Any OID with this prefix that is not explicitly mentioned in this document should be considered **unstable** and not used for any purpose.

9.1 Namespaces

OID Prefix: `1.3.6.1.4.1.53450`

OID Suffix	Description
`.1`	LDAP
`.1.1`	Controls
`.1.2`	Extensions
`.1.3`	Syntax rules
`.1.4`	Matching rules
`.1.5`	Attribute types
`.1.6`	Object classes

9.2 Objects

None currently assigned.

CHAPTER 10

Testing Setup

Warning: Testing has been moved to docker using public images. Check `.travis.yml` for details. This page is maintained for historical documentation purposes.

10.1 System

- Digital Ocean VPS with Debian 7.9
- OpenLDAP 2.4.31
- Cyrus SASL 2.1.25
- 389 Directory Server 1.3.6

10.2 SASL

10.2.1 SASL config ldif

```
dn: cn=config
changetype: modify
replace: olcAuthzRegexp
olcAuthzRegexp: uid=([^,]+),.* cn=$1,dc=example,dc=org
-
add: olcSaslAuxprops
olcSaslAuxprops: sasldb
-
add: olcSaslRealm
olcSaslRealm: example.org
-
add: olcSaslHost
```

(continues on next page)

(continued from previous page)

```
olcSaslHost: example.org
-
```

10.2.2 Adding sasl user password with

```
saslpasswd2 -u example.org -c $USER
```

10.2.3 SASL auth control test case

```
% ldapwhoami -Y DIGEST-MD5 -U admin -H ldap://127.0.0.1
SASL/DIGEST-MD5 authentication started
Please enter your password:
SASL username: admin
SASL SSF: 128
SASL data security layer installed.
dn:cn=admin,dc=example,dc=org
```

10.3 LDAPS/StartTLS

- Certs set up following this Stack Overflow answer.
- Configured OpenLDAP as follows:

```
dn: cn=config
changetype: modify
replace: olcTLSCertificateKeyFile
olcTLSCertificateKeyFile: /certs/serverkey.pem
-
replace: olcTLSCertificateFile
olcTLSCertificateFile: /certs/servercert.pem
-
replace: olcTLSCACertificateFile
olcTLSCACertificateFile: /certs/cacert.pem
```

- Added `ldaps://127.0.0.1:636` to `SLAPD_SERVICES` in `/etc/default/slapd`

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

laurelin, 37
laurelin.extensions, 35
laurelin.ldap.exceptions, 36

Index

A

Abandon, 36

C

ConnectionAlreadyBound, 36
ConnectionUnbound, 36

I

InvalidBindState, 36
InvalidSyntaxError, 36

L

laurelin (*module*), 37
laurelin.extensions (*module*), 35
laurelin.ldap.exceptions (*module*), 36
LDAPConnectionError, 36
LDAPError, 36
LDAPExtensionError, 36
LDAPSASLError, 36
LDAPSchemaError, 36
LDAPSupportError, 36
LDAPTransactionError, 36
LDAPUnicodeWarning, 36
LDAPUnsolicitedMessage, 36
LDAPValidationError, 37
LDAPWarning, 37

M

MultipleSearchResults, 37

N

NoSearchResults, 37

P

ProhibitedCharacterError, 37

T

TagError, 37

U

UnexpectedResponseType, 37
UnexpectedSearchResults, 37