django-auth-ldap Documentation

Release 2.1.1

Peter Sagerson

Mar 26, 2020

Contents

This is a Django authentication backend that authenticates against an LDAP service. Configuration can be as simple as a single distinguished name template, but there are many rich configuration options for working with users, groups, and permissions.

- Documentation: https://django-auth-ldap.readthedocs.io/
- PyPI: https://pypi.org/project/django-auth-ldap/
- Repository: https://github.com/django-auth-ldap/django-auth-ldap
- Tests: http://travis-ci.org/django-auth-ldap/django-auth-ldap
- License: BSD 2-Clause

This version is supported on Python 3.5+; and Django 1.11+. It requires python-ldap >= 3.1.

CHAPTER 1

Installation

Install the package with pip:

```
$ pip install django-auth-ldap
```

It requires python-ldap >= 3.0. You'll need the OpenLDAP libraries and headers available on your system.

To use the auth backend in a Django project, add `'django_auth_ldap.backend.LDAPBackend'` to `AUTHENTICATION_BACKENDS`. Do not add anything to `INSTALLED_APPS`.

```
AUTHENTICATION_BACKENDS = ["django_auth_ldap.backend.LDAPBackend"]
```

`LDAPBackend` should work with custom user models, but it does assume that a database is present.

Note: *`LDAPBackend`* does not inherit from `ModelBackend`. It is possible to use *`LDAPBackend`* exclusively by configuring it to draw group membership from the LDAP server. However, if you would like to assign permissions to individual users or add users to groups within Django, you'll need to have both backends installed:

```
AUTHENTICATION_BACKENDS = [
    "django_auth_ldap.backend.LDAPBackend",
    "django.contrib.auth.backends.ModelBackend",
]
```

CHAPTER 2

Authentication

2.1 Server Config

If your LDAP server isn't running locally on the default port, you'll want to start by setting *AUTH_LDAP_SERVER_URI* to point to your server. The value of this setting can be anything that your LDAP library supports. For instance, openldap may allow you to give a comma- or space-separated list of URIs to try in sequence.

```
AUTH_LDAP_SERVER_URI = "ldap://ldap.example.com"
```

If your server location is even more dynamic than this, you may provide a function (or any callable object) that returns the URI. The callable is passed a single positional argument: `request`. You should assume that this will be called on every request, so if it's an expensive operation, some caching is in order.

```
from my_module import find_my_ldap_server

AUTH_LDAP_SERVER_URI = find_my_ldap_server
```

If you need to configure any python-ldap options, you can set *AUTH_LDAP_GLOBAL_OPTIONS* and/or *AUTH_LDAP_CONNECTION_OPTIONS*. For example, disabling referrals is not uncommon:

```
import ldap

AUTH_LDAP_CONNECTION_OPTIONS = {ldap.OPT_REFERRALS: 0}
```

Changed in version 1.7.0: When `AUTH_LDAP_SERVER_URI` is set to a callable, it is now passed a positional `request` argument. Support for no arguments will continue for backwards compatibility but will be removed in a future version.

2.2 Search/Bind

Now that you can talk to your LDAP server, the next step is to authenticate a username and password. There are two ways to do this, called search/bind and direct bind. The first one involves connecting to the LDAP server either anonymously or with a fixed account and searching for the distinguished name of the authenticating user. Then we can attempt to bind again with the user's password. The second method is to derive the user's DN from his username and attempt to bind as the user directly.

Because LDAP searches appear elsewhere in the configuration, the *LDAPSearch* class is provided to encapsulate search information. In this case, the filter parameter should contain the placeholder `%(user)s`. A simple configuration for the search/bind approach looks like this (some defaults included for completeness):

```
import ldap
from django_auth_ldap.config import LDAPSearch

AUTH_LDAP_BIND_DN = ""
AUTH_LDAP_BIND_PASSWORD = ""
AUTH_LDAP_USER_SEARCH = LDAPSearch(
    "ou=users,dc=example,dc=com", ldap.SCOPE_SUBTREE, "(uid=%(user)s)"
)
```

This will perform an anonymous bind, search under `"ou=users,dc=example,dc=com"` for an object with a uid matching the user's name, and try to bind using that DN and the user's password. The search must return exactly one result or authentication will fail. If you can't search anonymously, you can set *AUTH_LDAP_BIND_DN* to the distinguished name of an authorized user and *AUTH_LDAP_BIND_PASSWORD* to the password.

2.2.1 Search Unions

New in version 1.1.

If you need to search in more than one place for a user, you can use *LDAPSearchUnion*. This takes multiple LDAPSearch objects and returns the union of the results. The precedence of the underlying searches is unspecified.

```
import ldap
from django_auth_ldap.config import LDAPSearch, LDAPSearchUnion

AUTH_LDAP_USER_SEARCH = LDAPSearchUnion(
    LDAPSearch("ou=users,dc=example,dc=com", ldap.SCOPE_SUBTREE, "(uid=%(user)s)"),
    LDAPSearch("ou=otherusers,dc=example,dc=com", ldap.SCOPE_SUBTREE, "(uid=%(user)s)
↪"),
)
```

2.3 Direct Bind

To skip the search phase, set *AUTH_LDAP_USER_DN_TEMPLATE* to a template that will produce the authenticating user's DN directly. This template should have one placeholder, `%(user)s`. If the first example had used `ldap.SCOPE_ONELEVEL`, the following would be a more straightforward (and efficient) equivalent:

```
AUTH_LDAP_USER_DN_TEMPLATE = "uid=%(user)s,ou=users,dc=example,dc=com"
```

2.4 Customizing Authentication

New in version 1.3.

It is possible to further customize the authentication process by subclassing *LDAPBackend* and overriding *authenticate_ldap_user()*. The first argument is the unauthenticated *ldap_user*, the second is the supplied password. The intent is to give subclasses a simple pre- and post-authentication hook.

If a subclass decides to proceed with the authentication, it must call the inherited implementation. It may then return either the authenticated user or None. The behavior of any other return value–such as substituting a different user object–is undefined. *User objects* has more on managing Django user objects.

Obviously, it is always safe to access ldap_user.dn before authenticating the user. Accessing ldap_user.attrs and others should be safe unless you're relying on special binding behavior, such as *AUTH_LDAP_BIND_AS_AUTHENTICATING_USER*.

2.5 Notes

LDAP is fairly flexible when it comes to matching DNs. *LDAPBackend* makes an effort to accommodate this by forcing usernames to lower case when creating Django users and trimming whitespace when authenticating.

Some LDAP servers are configured to allow users to bind without a password. As a precaution against false positives, *LDAPBackend* will summarily reject any authentication attempt with an empty password. You can disable this behavior by setting *AUTH_LDAP_PERMIT_EMPTY_PASSWORD* to True.

By default, all LDAP operations are performed with the *AUTH_LDAP_BIND_DN* and *AUTH_LDAP_BIND_PASSWORD* credentials, not with the user's. Otherwise, the LDAP connection would be bound as the authenticating user during login requests and as the default credentials during other requests, so you might see inconsistent LDAP attributes depending on the nature of the Django view. If you're willing to accept the inconsistency in order to retrieve attributes while bound as the authenticating user, see *AUTH_LDAP_BIND_AS_AUTHENTICATING_USER*.

By default, LDAP connections are unencrypted and make no attempt to protect sensitive information, such as passwords. When communicating with an LDAP server on localhost or on a local network, this might be fine. If you need a secure connection to the LDAP server, you can either use an ldaps:// URL or enable the StartTLS extension. The latter is generally the preferred mechanism. To enable StartTLS, set *AUTH_LDAP_START_TLS* to True:

```
AUTH_LDAP_START_TLS = True
```

If *LDAPBackend* receives an LDAPError from python_ldap, it will normally swallow it and log a warning. If you'd like to perform any special handling for these exceptions, you can add a signal handler to *django_auth_ldap.backend.ldap_error*. The signal handler can handle the exception any way you like, including re-raising it or any other exception.

CHAPTER 3

Working With Groups

3.1 Types of Groups

Working with groups in LDAP can be a tricky business, mostly because there are so many different kinds. This module includes an extensible API for working with any kind of group and includes implementations for the most common ones. `LDAPGroupType` is a base class whose concrete subclasses can determine group membership for particular grouping mechanisms. Four built-in subclasses cover most grouping mechanisms:

- `PosixGroupType`
- `MemberDNGroupType`
- `NestedMemberDNGroupType`

posixGroup and nisNetgroup objects are somewhat specialized, so they get their own classes. The other two cover mechanisms whereby a group object stores a list of its members as distinguished names. This includes groupOfNames, groupOfUniqueNames, and Active Directory groups, among others. The nested variant allows groups to contain other groups, to as many levels as you like. For convenience and readability, several trivial subclasses of the above are provided:

- `GroupOfNamesType`
- `NestedGroupOfNamesType`
- `GroupOfUniqueNamesType`
- `NestedGroupOfUniqueNamesType`
- `ActiveDirectoryGroupType`
- `NestedActiveDirectoryGroupType`
- `OrganizationalRoleGroupType`
- `NestedOrganizationalRoleGroupType`

3.2 Finding Groups

To get started, you'll need to provide some basic information about your LDAP groups. *AUTH_LDAP_GROUP_SEARCH* is an *LDAPSearch* object that identifies the set of relevant group objects. That is, all groups that users might belong to as well as any others that we might need to know about (in the case of nested groups, for example). *AUTH_LDAP_GROUP_TYPE* is an instance of the class corresponding to the type of group that will be returned by *AUTH_LDAP_GROUP_SEARCH*. All groups referenced elsewhere in the configuration must be of this type and part of the search results.

```
import ldap
from django_auth_ldap.config import LDAPSearch, GroupOfNamesType

AUTH_LDAP_GROUP_SEARCH = LDAPSearch(
    "ou=groups,dc=example,dc=com", ldap.SCOPE_SUBTREE, "(objectClass=groupOfNames)"
)
AUTH_LDAP_GROUP_TYPE = GroupOfNamesType()
```

3.3 Limiting Access

The simplest use of groups is to limit the users who are allowed to log in. If *AUTH_LDAP_REQUIRE_GROUP* is set, then only users who are members of that group will successfully authenticate. *AUTH_LDAP_DENY_GROUP* is the reverse: if given, members of this group will be rejected.

```
AUTH_LDAP_REQUIRE_GROUP = "cn=enabled,ou=groups,dc=example,dc=com"
AUTH_LDAP_DENY_GROUP = "cn=disabled,ou=groups,dc=example,dc=com"
```

However, these two settings alone may not be enough to satisfy your needs. In such cases, you can use the *LDAPGroupQuery* object to perform more complex matches against a user's groups. For example:

```
from django_auth_ldap.config import LDAPGroupQuery

AUTH_LDAP_REQUIRE_GROUP = (
    LDAPGroupQuery("cn=enabled,ou=groups,dc=example,dc=com")
    | LDAPGroupQuery("cn=also_enabled,ou=groups,dc=example,dc=com")
) & ~LDAPGroupQuery("cn=disabled,ou=groups,dc=example,dc=com")
```

It is important to note a couple features of the example above. First and foremost, this handles the case of both *AUTH_LDAP_REQUIRE_GROUP* and *AUTH_LDAP_DENY_GROUP* in one setting. Second, you can use three operators on these queries: `&`, `|`, and `~`: `and`, `or`, and `not`, respectively.

When groups are configured, you can always get the list of a user's groups from `user.ldap_user.group_dns` or `user.ldap_user.group_names`. More advanced uses of groups are covered in the next two sections.

CHAPTER 4

User objects

Authenticating against an external source is swell, but Django's auth module is tightly bound to a user model. When a user logs in, we have to create a model object to represent them in the database. Because the LDAP search is case-insensitive, the default implementation also searches for existing Django users with an iexact query and new users are created with lowercase usernames. See *`get_or_build_user()`* if you'd like to override this behavior. See *`get_user_model()`* if you'd like to substitute a proxy model.

By default, lookups on existing users are done using the user model's `USERNAME_FIELD`. To lookup by a different field, use *`AUTH_LDAP_USER_QUERY_FIELD`*. When set, the username field is ignored.

When using the default for lookups, the only required field for a user is the username. The default `User` model can be picky about the characters allowed in usernames, so *`LDAPBackend`* includes a pair of hooks, *`ldap_to_django_username()`* and *`django_to_ldap_username()`*, to translate between LDAP usernames and Django usernames. You may need this, for example, if your LDAP names have periods in them. You can subclass *`LDAPBackend`* to implement these hooks; by default the username is not modified. `User` objects that are authenticated by *`LDAPBackend`* will have an `ldap_username` attribute with the original (LDAP) username. `username` (or `get_username()`) will, of course, be the Django username.

Note: Users created by *`LDAPBackend`* will have an unusable password set. This will only happen when the user is created, so if you set a valid password in Django, the user will be able to log in through `ModelBackend` (if configured) even if they are rejected by LDAP. This is not generally recommended, but could be useful as a fail-safe for selected users in case the LDAP server is unavailable.

4.1 Populating Users

You can perform arbitrary population of your user models by adding listeners to the `Django signal`: *`django_auth_ldap.backend.populate_user`*. This signal is sent after the user object has been constructed (but not necessarily saved) and any configured attribute mapping has been applied (see below). You can use this to propagate information from the LDAP directory to the user object any way you like. If you need the user object to exist in the database at this point, you can save it in your signal handler or override *`get_or_build_user()`*. In either case, the user instance will be saved automatically after the signal handlers are run.

If you need an attribute that isn't included by default in the LDAP search results, see *AUTH_LDAP_USER_ATTRLIST*.

4.2 Easy Attributes

If you just want to copy a few attribute values directly from the user's LDAP directory entry to their Django user, the setting, *AUTH_LDAP_USER_ATTR_MAP*, makes it easy. This is a dictionary that maps user model keys, respectively, to (case-insensitive) LDAP attribute names:

```
AUTH_LDAP_USER_ATTR_MAP = {"first_name": "givenName", "last_name": "sn"}
```

Only string fields can be mapped to attributes. Boolean fields can be defined by group membership:

```
AUTH_LDAP_USER_FLAGS_BY_GROUP = {
    "is_active": "cn=active,ou=groups,dc=example,dc=com",
    "is_staff": (
        LDAPGroupQuery("cn=staff,ou=groups,dc=example,dc=com")
        | LDAPGroupQuery("cn=admin,ou=groups,dc=example,dc=com")
    ),
    "is_superuser": "cn=superuser,ou=groups,dc=example,dc=com",
}
```

Values in this dictionary may be simple DNs (as strings), lists or tuples of DNs, or *LDAPGroupQuery* instances. Lists are converted to queries joined by `|`.

Remember that if these settings don't do quite what you want, you can always use the signals described in the previous section to implement your own logic.

4.3 Updating Users

By default, all mapped user fields will be updated each time the user logs in. To disable this, set *AUTH_LDAP_ALWAYS_UPDATE_USER* to `False`. If you need to populate a user outside of the authentication process—for example, to create associated model objects before the user logs in for the first time—you can call *django_auth_ldap.backend.LDAPBackend.populate_user()*. You'll need an instance of *LDAPBackend*, which you should feel free to create yourself. *populate_user()* returns the `User` or *None* if the user could not be found in LDAP.

```
from django_auth_ldap.backend import LDAPBackend

user = LDAPBackend().populate_user("alice")
if user is None:
    raise Exception("No user named alice")
```

4.4 Direct Attribute Access

If you need to access multi-value attributes or there is some other reason that the above is inadequate, you can also access the user's raw LDAP attributes. `user.ldap_user` is an object with four public properties. The group properties are, of course, only valid if groups are configured.

- `dn`: The user's distinguished name.

- `attrs`: The user's LDAP attributes as a dictionary of lists of string values. The dictionaries are modified to use case-insensitive keys.
- `group_dns`: The set of groups that this user belongs to, as DNs.
- `group_names`: The set of groups that this user belongs to, as simple names. These are the names that will be used if *`AUTH_LDAP_MIRROR_GROUPS`* is used.

Python-ldap returns all attribute values as utf8-encoded strings. For convenience, this module will try to decode all values into Unicode strings. Any string that can not be successfully decoded will be left as-is; this may apply to binary values such as Active Directory's objectSid.

CHAPTER 5

Permissions

Groups are useful for more than just populating the user's `is_*` fields. *LDAPBackend* would not be complete without some way to turn a user's LDAP group memberships into Django model permissions. In fact, there are two ways to do this.

Ultimately, both mechanisms need some way to map LDAP groups to Django groups. Implementations of *LDAPGroupType* will have an algorithm for deriving the Django group name from the LDAP group. Clients that need to modify this behavior can subclass the *LDAPGroupType* class. All of the built-in implementations take a `name_attr` argument to `__init__`, which specifies the LDAP attribute from which to take the Django group name. By default, the `cn` attribute is used.

5.1 Using Groups Directly

The least invasive way to map group permissions is to set *AUTH_LDAP_FIND_GROUP_PERMS* to `True`. *LDAPBackend* will then find all of the LDAP groups that a user belongs to, map them to Django groups, and load the permissions for those groups. You will need to create the Django groups and associate permissions yourself, generally through the admin interface.

To minimize traffic to the LDAP server, *LDAPBackend* can make use of Django's cache framework to keep a copy of a user's LDAP group memberships. To enable this feature, set *AUTH_LDAP_CACHE_TIMEOUT*, which determines the timeout of cache entries in seconds.

```
AUTH_LDAP_CACHE_TIMEOUT = 3600
```

5.2 Group Mirroring

The second way to turn LDAP group memberships into permissions is to mirror the groups themselves. This approach has some important disadvantages and should be avoided if possible. For one thing, membership will only be updated when the user authenticates, which may be especially inappropriate for sites with long session timeouts.

If *AUTH_LDAP_MIRROR_GROUPS* is `True`, then every time a user logs in, *LDAPBackend* will update the database with the user's LDAP groups. Any group that doesn't exist will be created and the user's Django group membership will be updated to exactly match their LDAP group membership. If the LDAP server has nested groups, the Django database will end up with a flattened representation. For group mirroring to have any effect, you of course need `ModelBackend` installed as an authentication backend.

By default, we assume that LDAP is the sole authority on group membership; if you remove a user from a group in LDAP, they will be removed from the corresponding Django group the next time they log in. It is also possible to have django-auth-ldap ignore some Django groups, presumably because they are managed manually or through some other mechanism. If *AUTH_LDAP_MIRROR_GROUPS* is a list of group names, we will manage these groups and no others. If *AUTH_LDAP_MIRROR_GROUPS_EXCEPT* is a list of group names, we will manage all groups except those named; *AUTH_LDAP_MIRROR_GROUPS* is ignored in this case.

5.3 Non-LDAP Users

LDAPBackend has one more feature pertaining to permissions, which is the ability to handle authorization for users that it did not authenticate. For example, you might be using `RemoteUserBackend` to map externally authenticated users to Django users. By setting *AUTH_LDAP_AUTHORIZE_ALL_USERS*, *LDAPBackend* will map these users to LDAP users in the normal way in order to provide authorization information. Note that this does *not* work with *AUTH_LDAP_MIRROR_GROUPS*; group mirroring is a feature of authentication, not authorization.

CHAPTER 6

Multiple LDAP Configs

New in version 1.1.

You've probably noticed that all of the settings for this backend have the prefix AUTH_LDAP_. This is the default, but it can be customized by subclasses of *LDAPBackend*. The main reason you would want to do this is to create two backend subclasses that reference different collections of settings and thus operate independently. For example, you might have two separate LDAP servers that you want to authenticate against. A short example should demonstrate this:

```
# mypackage.ldap

from django_auth_ldap.backend import LDAPBackend


class LDAPBackend1(LDAPBackend):
    settings_prefix = "AUTH_LDAP_1_"


class LDAPBackend2(LDAPBackend):
    settings_prefix = "AUTH_LDAP_2_"
```

```
# settings.py

AUTH_LDAP_1_SERVER_URI = "ldap://ldap1.example.com"
AUTH_LDAP_1_USER_DN_TEMPLATE = "uid=%(user)s,ou=users,dc=example,dc=com"

AUTH_LDAP_2_SERVER_URI = "ldap://ldap2.example.com"
AUTH_LDAP_2_USER_DN_TEMPLATE = "uid=%(user)s,ou=users,dc=example,dc=com"

AUTHENTICATION_BACKENDS = ("mypackage.ldap.LDAPBackend1", "mypackage.ldap.LDAPBackend2
↪")
```

All of the usual rules apply: Django will attempt to authenticate a user with each backend in turn until one of them succeeds. When a particular backend successfully authenticates a user, that user will be linked to the backend for the duration of their session.

Note: Due to its global nature, *AUTH_LDAP_GLOBAL_OPTIONS* ignores the settings prefix. Regardless of how many backends are installed, this setting is referenced once by its default name at the time we load the ldap module.

CHAPTER 7

Custom Behavior

There are times that the default *LDAPBackend* behavior may be insufficient for your needs. In those cases, you can further customize the behavior by following these general steps:

- Create your own *LDAPBackend* subclass.
- Use *default_settings* to define any custom settings you may want to use.
- Override *authenticate_ldap_user()* hook and/or any other method as needed.
- Define additional methods and attributes as needed.
- Access your custom settings via `self.settings` inside your *LDAPBackend* subclass.

7.1 Subclassing LDAPBackend

You can implement your own *LDAPBackend* subclass if you need some custom behavior. For example, you want to only allow 50 login attempts every 30 minutes, and those numbers may change as needed. Furthermore, any successful login attempt against the LDAP server must send out an SMS notification, but there should be an option to limit this behavior to a specific set of usernames based on a regex. One can accomplish that by doing something like this:

```
# mypackage.ldap

import re

from django.core.cache import cache

from django_auth_ldap.backend import LDAPBackend


class CustomLDAPBackend(LDAPBackend):
    default_settings = {
        "LOGIN_COUNTER_KEY": "CUSTOM_LDAP_LOGIN_ATTEMPT_COUNT",
        "LOGIN_ATTEMPT_LIMIT": 50,
        "RESET_TIME": 30 * 60,
```

(continues on next page)

(continued from previous page)

```
        "USERNAME_REGEX": r"^.*$",
    }

    def authenticate_ldap_user(self, ldap_user, password):
        if self.exceeded_login_attempt_limit():
            # Or you can raise a 403 if you do not want
            # to continue checking other auth backends
            print("Login attempts exceeded.")
            return None
        self.increment_login_attempt_count()
        user = ldap_user.authenticate(password)
        if user and self.username_matches_regex(user.username):
            self.send_sms(user.username)
        return user

    @property
    def login_attempt_count(self):
        return cache.get_or_set(
            self.settings.LOGIN_COUNTER_KEY, 0, self.settings.RESET_TIME
        )

    def increment_login_attempt_count(self):
        try:
            cache.incr(self.settings.LOGIN_COUNTER_KEY)
        except ValueError:
            cache.set(self.settings.LOGIN_COUNTER_KEY, 1, self.settings.RESET_TIME)

    def exceeded_login_attempt_limit(self):
        return self.login_attempt_count >= self.settings.LOGIN_ATTEMPT_LIMIT

    def username_matches_regex(self, username):
        return re.match(self.settings.USERNAME_REGEX, username)

    def send_sms(self, username):
        # Implement your SMS logic here
        print("SMS sent!")
```

```
# settings.py

AUTHENTICATION_BACKENDS = [
    # ...
    "mypackage.ldap.CustomLDAPBackend",
    # ...
]
```

7.2 Using default_settings

While you can use your own custom Django settings to create something similar to the sample code above, there are a couple of advantages in using `default_settings` instead.

Following the sample code above, one advantage is that the subclass will now automatically check your Django settings for `AUTH_LDAP_LOGIN_COUNTER_KEY`, `AUTH_LDAP_LOGIN_ATTEMPT_LIMIT`, `AUTH_LDAP_RESET_TIME`, and `AUTH_LDAP_USERNAME_REGEX`. Another advantage is that for each setting not explicitly defined in your Django settings, the subclass will then use the corresponding default values. This behavior will be very handy in case you will need to override certain settings.

7.3 Overriding default_settings

If down the line, you want to increase the login attempt limit to 100 every 15 minutes, and you only want SMS notifications for usernames with a "zz_" prefix, then you can simply modify your settings.py like so.

```
# settings.py

AUTH_LDAP_LOGIN_ATTEMPT_LIMIT = 100
AUTH_LDAP_RESET_TIME = 15 * 60
AUTH_LDAP_USERNAME_REGEX = r"^zz_.*$"

AUTHENTICATION_BACKENDS = [
    # ...
    "mypackage.ldap.CustomLDAPBackend",
    # ...
]
```

If the `settings_prefix` of the subclass was also changed, then the prefix must also be used in your settings. For example, if the prefix was changed to "AUTH_LDAP_1_", then it should look like this.

```
# settings.py

AUTH_LDAP_1_LOGIN_ATTEMPT_LIMIT = 100
AUTH_LDAP_1_RESET_TIME = 15 * 60
AUTH_LDAP_1_USERNAME_REGEX = r"^zz_.*$"

AUTHENTICATION_BACKENDS = [
    # ...
    "mypackage.ldap.CustomLDAPBackend",
    # ...
]
```

CHAPTER 8

Logging

LDAPBackend uses the standard Python logging module to log debug and warning messages to the logger named 'django_auth_ldap'. If you need debug messages to help with configuration issues, you should add a handler to this logger. Using Django's LOGGING setting, you can add an entry to your config.

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {"console": {"class": "logging.StreamHandler"}},
    "loggers": {"django_auth_ldap": {"level": "DEBUG", "handlers": ["console"]}},
}
```

CHAPTER 9

Performance

LDAPBackend is carefully designed not to require a connection to the LDAP service for every request. Of course, this depends heavily on how it is configured. If LDAP traffic or latency is a concern for your deployment, this section has a few tips on minimizing it, in decreasing order of impact.

1. **Cache groups**. If AUTH_LDAP_FIND_GROUP_PERMS is True, the default behavior is to reload a user's group memberships on every request. This is the safest behavior, as any membership change takes effect immediately, but it is expensive. If possible, set AUTH_LDAP_CACHE_TIMEOUT to remove most of this traffic.
2. **Don't access user.ldap_user.***. Except for ldap_user.dn, these properties are only cached on a per-request basis. If you can propagate LDAP attributes to a User, they will only be updated at login. user.ldap_user.attrs triggers an LDAP connection for every request in which it's accessed.
3. **Use simpler group types**. Some grouping mechanisms are more expensive than others. This will often be outside your control, but it's important to note that the extra functionality of more complex group types like NestedGroupOfNamesType is not free and will generally require a greater number and complexity of LDAP queries.
4. **Use direct binding**. Binding with AUTH_LDAP_USER_DN_TEMPLATE is a little bit more efficient than relying on AUTH_LDAP_USER_SEARCH. Specifically, it saves two LDAP operations (one bind and one search) per login.

CHAPTER 10

Example Configuration

Here is a complete example configuration from `settings.py` that exercises nearly all of the features. In this example, we're authenticating against a global pool of users in the directory, but we have a special area set aside for Django groups (`ou=django,ou=groups,dc=example,dc=com`). Remember that most of this is optional if you just need simple authentication. Some default settings and arguments are included for completeness.

```
import ldap
from django_auth_ldap.config import LDAPSearch, GroupOfNamesType


# Baseline configuration.
AUTH_LDAP_SERVER_URI = "ldap://ldap.example.com"

AUTH_LDAP_BIND_DN = "cn=django-agent,dc=example,dc=com"
AUTH_LDAP_BIND_PASSWORD = "phlebotinum"
AUTH_LDAP_USER_SEARCH = LDAPSearch(
    "ou=users,dc=example,dc=com", ldap.SCOPE_SUBTREE, "(uid=%(user)s)"
)
# Or:
# AUTH_LDAP_USER_DN_TEMPLATE = 'uid=%(user)s,ou=users,dc=example,dc=com'

# Set up the basic group parameters.
AUTH_LDAP_GROUP_SEARCH = LDAPSearch(
    "ou=django,ou=groups,dc=example,dc=com",
    ldap.SCOPE_SUBTREE,
    "(objectClass=groupOfNames)",
)
AUTH_LDAP_GROUP_TYPE = GroupOfNamesType(name_attr="cn")

# Simple group restrictions
AUTH_LDAP_REQUIRE_GROUP = "cn=enabled,ou=django,ou=groups,dc=example,dc=com"
AUTH_LDAP_DENY_GROUP = "cn=disabled,ou=django,ou=groups,dc=example,dc=com"

# Populate the Django user from the LDAP directory.
AUTH_LDAP_USER_ATTR_MAP = {
```

(continues on next page)

(continued from previous page)

```
    "first_name": "givenName",
    "last_name": "sn",
    "email": "mail",
}

AUTH_LDAP_USER_FLAGS_BY_GROUP = {
    "is_active": "cn=active,ou=django,ou=groups,dc=example,dc=com",
    "is_staff": "cn=staff,ou=django,ou=groups,dc=example,dc=com",
    "is_superuser": "cn=superuser,ou=django,ou=groups,dc=example,dc=com",
}

# This is the default, but I like to be explicit.
AUTH_LDAP_ALWAYS_UPDATE_USER = True

# Use LDAP group membership to calculate group permissions.
AUTH_LDAP_FIND_GROUP_PERMS = True

# Cache distinguished names and group memberships for an hour to minimize
# LDAP traffic.
AUTH_LDAP_CACHE_TIMEOUT = 3600

# Keep ModelBackend around for per-user permissions and maybe a local
# superuser.
AUTHENTICATION_BACKENDS = (
    "django_auth_ldap.backend.LDAPBackend",
    "django.contrib.auth.backends.ModelBackend",
)
```

CHAPTER 11

Reference

11.1 Settings

11.1.1 AUTH_LDAP_ALWAYS_UPDATE_USER

Default: `True`

If `True`, the fields of a `User` object will be updated with the latest values from the LDAP directory every time the user logs in. Otherwise the `User` object will only be populated when it is automatically created.

11.1.2 AUTH_LDAP_AUTHORIZE_ALL_USERS

Default: `False`

If `True`, *`LDAPBackend`* will be able furnish permissions for any Django user, regardless of which backend authenticated it.

11.1.3 AUTH_LDAP_BIND_AS_AUTHENTICATING_USER

Default: `False`

If `True`, authentication will leave the LDAP connection bound as the authenticating user, rather than forcing it to re-bind with the default credentials after authentication succeeds. This may be desirable if you do not have global credentials that are able to access the user's attributes. django-auth-ldap never stores the user's password, so this only applies to requests where the user is authenticated. Thus, the downside to this setting is that LDAP results may vary based on whether the user was authenticated earlier in the Django view, which could be surprising to code not directly concerned with authentication.

11.1.4 AUTH_LDAP_BIND_DN

Default: `''` (Empty string)

The distinguished name to use when binding to the LDAP server (with *AUTH_LDAP_BIND_PASSWORD*). Use the empty string (the default) for an anonymous bind. To authenticate a user, we will bind with that user's DN and password, but for all other LDAP operations, we will be bound as the DN in this setting. For example, if *AUTH_LDAP_USER_DN_TEMPLATE* is not set, we'll use this to search for the user. If *AUTH_LDAP_FIND_GROUP_PERMS* is `True`, we'll also use it to determine group membership.

11.1.5 AUTH_LDAP_BIND_PASSWORD

Default: `''` (Empty string)

The password to use with *AUTH_LDAP_BIND_DN*.

11.1.6 AUTH_LDAP_CACHE_TIMEOUT

Default: `0`

The value determines the amount of time, in seconds, a user's group memberships and distinguished name are cached. The value `0`, the default, disables caching entirely.

Changed in version 1.6.0: Previously caching was controlled by the settings *AUTH_LDAP_CACHE_GROUPS* and *AUTH_LDAP_GROUP_CACHE_TIMEOUT*. If *AUTH_LDAP_CACHE_GROUPS* is set, the *AUTH_LDAP_CACHE_TIMEOUT* value is derievd from these deprecated settings.

11.1.7 AUTH_LDAP_CONNECTION_OPTIONS

Default: `{}`

A dictionary of options to pass to each connection to the LDAP server via `LDAPObject.set_option()`. Keys are ldap.OPT_* constants.

11.1.8 AUTH_LDAP_DENY_GROUP

Default: `None`

The distinguished name of a group; authentication will fail for any user that belongs to this group.

11.1.9 AUTH_LDAP_FIND_GROUP_PERMS

Default: `False`

If `True`, *LDAPBackend* will furnish group permissions based on the LDAP groups the authenticated user belongs to. *AUTH_LDAP_GROUP_SEARCH* and *AUTH_LDAP_GROUP_TYPE* must also be set.

11.1.10 AUTH_LDAP_GLOBAL_OPTIONS

Default: `{}`

A dictionary of options to pass to `ldap.set_option()`. Keys are ldap.OPT_* constants.

Note: Due to its global nature, this setting ignores the *settings prefix*. Regardless of how many backends are installed, this setting is referenced once by its default name at the time we load the ldap module.

11.1.11 AUTH_LDAP_GROUP_SEARCH

Default: `None`

An *`LDAPSearch`* object that finds all LDAP groups that users might belong to. If your configuration makes any references to LDAP groups, this and *`AUTH_LDAP_GROUP_TYPE`* must be set.

11.1.12 AUTH_LDAP_GROUP_TYPE

Default: `None`

An *`LDAPGroupType`* instance describing the type of group returned by *`AUTH_LDAP_GROUP_SEARCH`*.

11.1.13 AUTH_LDAP_MIRROR_GROUPS

Default: `None`

If `True`, *`LDAPBackend`* will mirror a user's LDAP group membership in the Django database. Any time a user authenticates, we will create all of their LDAP groups as Django groups and update their Django group membership to exactly match their LDAP group membership. If the LDAP server has nested groups, the Django database will end up with a flattened representation.

This can also be a list or other collection of group names, in which case we'll only mirror those groups and leave the rest alone. This is ignored if *`AUTH_LDAP_MIRROR_GROUPS_EXCEPT`* is set.

11.1.14 AUTH_LDAP_MIRROR_GROUPS_EXCEPT

Default: `None`

If this is not `None`, it must be a list or other collection of group names. This will enable group mirroring, except that we'll never change the membership of the indicated groups. *`AUTH_LDAP_MIRROR_GROUPS`* is ignored in this case.

11.1.15 AUTH_LDAP_PERMIT_EMPTY_PASSWORD

Default: `False`

If `False` (the default), authentication with an empty password will fail immediately, without any LDAP communication. This is a secure default, as some LDAP servers are configured to allow binds to succeed with no password, perhaps at a reduced level of access. If you need to make use of this LDAP feature, you can change this setting to `True`.

11.1.16 AUTH_LDAP_REQUIRE_GROUP

Default: `None`

The distinguished name of a group; authentication will fail for any user that does not belong to this group. This can also be an *`LDAPGroupQuery`* instance.

11.1.17 AUTH_LDAP_NO_NEW_USERS

Default: `False`

Prevent the creation of new users during authentication. Any users not already in the Django user database will not be able to login.

11.1.18 AUTH_LDAP_SERVER_URI

Default: `'ldap://localhost'`

The URI of the LDAP server. This can be any URI that is supported by your underlying LDAP libraries. Can also be a callable that returns the URI. The callable is passed a single positional argument: `request`.

Changed in version 1.7.0: When `AUTH_LDAP_SERVER_URI` is set to a callable, it is now passed a positional `request` argument. Support for no arguments will continue for backwards compatibility but will be removed in a future version.

11.1.19 AUTH_LDAP_START_TLS

Default: `False`

If `True`, each connection to the LDAP server will call `start_tls_s()` to enable TLS encryption over the standard LDAP port. There are a number of configuration options that can be given to *`AUTH_LDAP_GLOBAL_OPTIONS`* that affect the TLS connection. For example, `ldap.OPT_X_TLS_REQUIRE_CERT` can be set to `ldap.OPT_X_TLS_NEVER` to disable certificate verification, perhaps to allow self-signed certificates.

11.1.20 AUTH_LDAP_USER_QUERY_FIELD

Default: `None`

The field on the user model used to query the authenticating user in the database. If unset, uses the value of `USERNAME_FIELD` of the model class. When set, the value used to query is obtained through the *`AUTH_LDAP_USER_ATTR_MAP`*. For example, setting *`AUTH_LDAP_USER_QUERY_FIELD`* to `username` and adding `"username": "sAMAccountName",` to *`AUTH_LDAP_USER_ATTR_MAP`* will cause django to query local database using `username` column and LDAP using `sAMAccountName` attribute.

11.1.21 AUTH_LDAP_USER_ATTRLIST

Default: `None`

A list of attribute names to load for the authenticated user. Normally, you can ignore this and the LDAP server will send back all of the attributes of the directory entry. One reason you might need to override this is to get operational attributes, which are not normally included:

```
AUTH_LDAP_USER_ATTRLIST = ["*", "+"]
```

11.1.22 AUTH_LDAP_USER_ATTR_MAP

Default: `{}`

A mapping from `User` field names to LDAP attribute names. A users's `User` object will be populated from his LDAP attributes at login.

11.1.23 AUTH_LDAP_USER_DN_TEMPLATE

Default: `None`

A string template that describes any user's distinguished name based on the username. This must contain the placeholder `%(user)s`.

11.1.24 AUTH_LDAP_USER_FLAGS_BY_GROUP

Default: `{}`

A mapping from boolean `User` field names to distinguished names of LDAP groups. The corresponding field is set to `True` or `False` according to whether the user is a member of the group.

Values may be strings for simple group membership tests or *`LDAPGroupQuery`* instances for more complex cases.

11.1.25 AUTH_LDAP_USER_SEARCH

Default: `None`

An *`LDAPSearch`* object that will locate a user in the directory. The filter parameter should contain the placeholder `%(user)s` for the username. It must return exactly one result for authentication to succeed.

11.2 Module Properties

`django_auth_ldap.`**`version`**
The library's current version number as a 3-tuple.

`django_auth_ldap.`**`version_string`**
The library's current version number as a string.

11.3 Configuration

`class` `django_auth_ldap.config.`**`LDAPSearch`**

`__init__`(*base_dn*, *scope*, *filterstr='(objectClass=*)'*)

Parameters

- **`base_dn`** (*`str`*) – The distinguished name of the search base.
- **`scope`** (*`int`*) – One of `ldap.SCOPE_*`.
- **`filterstr`** (*`str`*) – An optional filter string (e.g. '(objectClass=person)'). In order to be valid, `filterstr` must be enclosed in parentheses.

`class` `django_auth_ldap.config.`**`LDAPSearchUnion`**
New in version 1.1.

`__init__`(**searches*)

Parameters **`searches`** (*`LDAPSearch`*) – Zero or more LDAPSearch objects. The result of the overall search is the union (by DN) of the results of the underlying searches. The precedence of the underlying results and the ordering of the final results are both undefined.

class `django_auth_ldap.config.`**`LDAPGroupType`**

The base class for objects that will determine group membership for various LDAP grouping mechanisms. Implementations are provided for common group types or you can write your own. See the source code for subclassing notes.

`__init__`(*name_attr='cn'*)

By default, LDAP groups will be mapped to Django groups by taking the first value of the cn attribute. You can specify a different attribute with `name_attr`.

class `django_auth_ldap.config.`**`PosixGroupType`**

A concrete subclass of *`LDAPGroupType`* that handles the `posixGroup` object class. This checks for both primary group and group membership.

`__init__`(*name_attr='cn'*)

class `django_auth_ldap.config.`**`MemberDNGroupType`**

A concrete subclass of *`LDAPGroupType`* that handles grouping mechanisms wherein the group object contains a list of its member DNs.

`__init__`(*member_attr*, *name_attr='cn'*)

Parameters **`member_attr`** (*`str`*) – The attribute on the group object that contains a list of member DNs. 'member' and 'uniqueMember' are common examples.

class `django_auth_ldap.config.`**`NestedMemberDNGroupType`**

Similar to *`MemberDNGroupType`*, except this allows groups to contain other groups as members. Group hierarchies will be traversed to determine membership.

`__init__`(*member_attr*, *name_attr='cn'*)

As above.

class `django_auth_ldap.config.`**`GroupOfNamesType`**

A concrete subclass of *`MemberDNGroupType`* that handles the `groupOfNames` object class. Equivalent to `MemberDNGroupType('member')`.

`__init__`(*name_attr='cn'*)

class `django_auth_ldap.config.`**`NestedGroupOfNamesType`**

A concrete subclass of *`NestedMemberDNGroupType`* that handles the `groupOfNames` object class. Equivalent to `NestedMemberDNGroupType('member')`.

`__init__`(*name_attr='cn'*)

class `django_auth_ldap.config.`**`GroupOfUniqueNamesType`**

A concrete subclass of *`MemberDNGroupType`* that handles the `groupOfUniqueNames` object class. Equivalent to `MemberDNGroupType('uniqueMember')`.

`__init__`(*name_attr='cn'*)

class `django_auth_ldap.config.`**`NestedGroupOfUniqueNamesType`**

A concrete subclass of *`NestedMemberDNGroupType`* that handles the `groupOfUniqueNames` object class. Equivalent to `NestedMemberDNGroupType('uniqueMember')`.

`__init__`(*name_attr='cn'*)

class `django_auth_ldap.config.`**`ActiveDirectoryGroupType`**

A concrete subclass of *`MemberDNGroupType`* that handles Active Directory groups. Equivalent to `MemberDNGroupType('member')`.

`__init__`(*name_attr='cn'*)

class `django_auth_ldap.config.`**`NestedActiveDirectoryGroupType`**

A concrete subclass of *`NestedMemberDNGroupType`* that handles Active Directory groups. Equivalent to `NestedMemberDNGroupType('member')`.

`__init__`(*name_attr='cn'*)

class `django_auth_ldap.config.`**`OrganizationalRoleGroupType`**

A concrete subclass of *`MemberDNGroupType`* that handles the `organizationalRole` object class. Equivalent to `MemberDNGroupType('roleOccupant')`.

`__init__`(*name_attr='cn'*)

class `django_auth_ldap.config.`**`NestedOrganizationalRoleGroupType`**

A concrete subclass of *`NestedMemberDNGroupType`* that handles the `organizationalRole` object class. Equivalent to `NestedMemberDNGroupType('roleOccupant')`.

`__init__`(*name_attr='cn'*)

class `django_auth_ldap.config.`**`LDAPGroupQuery`**

Represents a compound query for group membership.

This can be used to construct an arbitrarily complex group membership query with AND, OR, and NOT logical operators. Construct primitive queries with a group DN as the only argument. These queries can then be combined with the `&`, `|`, and `~` operators.

This is used by certain settings, including *`AUTH_LDAP_REQUIRE_GROUP`* and *`AUTH_LDAP_USER_FLAGS_BY_GROUP`*. An example is shown in *Limiting Access*.

`__init__`(*group_dn*)

Parameters **`group_dn`** (*`str`*) – The distinguished name of a group to test for membership.

11.4 Backend

`django_auth_ldap.backend.`**`populate_user`**

This is a Django signal that is sent when clients should perform additional customization of a `User` object. It is sent after a user has been authenticated and the backend has finished populating it, and just before it is saved. The client may take this opportunity to populate additional model fields, perhaps based on `ldap_user.attrs`. This signal has two keyword arguments: `user` is the `User` object and `ldap_user` is the same as `user.ldap_user`. The sender is the *`LDAPBackend`* class.

`django_auth_ldap.backend.`**`ldap_error`**

This is a Django signal that is sent when we receive an `ldap.LDAPError` exception. The signal has three keyword arguments:

- `context`: one of `'authenticate'`, `'get_group_permissions'`, or `'populate_user'`, indicating which API was being called when the exception was caught.
- `user`: the Django user being processed (if available).
- `exception`: the `LDAPError` object itself.

The sender is the *`LDAPBackend`* class (or subclass).

class `django_auth_ldap.backend.`**`LDAPBackend`**

`LDAPBackend` has one method that may be called directly and several that may be overridden in subclasses.

`settings_prefix`

A prefix for all of our Django settings. By default, this is `'AUTH_LDAP_'`, but subclasses can override this. When different subclasses use different prefixes, they can both be installed and operate independently.

default_settings
A dictionary of default settings. This is empty in *LDAPBackend*, but subclasses can populate this with values that will override the built-in defaults. Note that the keys should omit the `'AUTH_LDAP_'` prefix.

populate_user(*username*)
Populates the Django user for the given LDAP username. This connects to the LDAP directory with the default credentials and attempts to populate the indicated Django user as if they had just logged in. *AUTH_LDAP_ALWAYS_UPDATE_USER* is ignored (assumed `True`).

get_user_model(*self*)
Returns the user model that *get_or_build_user()* will instantiate. By default, custom user models will be respected. Subclasses would most likely override this in order to substitute a proxy model.

authenticate_ldap_user(*self*, *ldap_user*, *password*)
Given an LDAP user object and password, authenticates the user and returns a Django user object. See *Customizing Authentication*.

get_or_build_user(*self*, *username*, *ldap_user*)
Given a username and an LDAP user object, this must return a valid Django user model instance. The `username` argument has already been passed through *ldap_to_django_username()*. You can get information about the LDAP user via `ldap_user.dn` and `ldap_user.attrs`. The return value must be an (instance, created) two-tuple. The instance does not need to be saved.

The default implementation looks for the username with a case-insensitive query; if it's not found, the model returned by *get_user_model()* will be created with the lowercased username. New users will not be saved to the database until after the *django_auth_ldap.backend.populate_user* signal has been sent.

A subclass may override this to associate LDAP users to Django users any way it likes.

ldap_to_django_username(*username*)
Returns a valid Django username based on the given LDAP username (which is what the user enters). By default, `username` is returned unchanged. This can be overridden by subclasses.

django_to_ldap_username(*username*)
The inverse of *ldap_to_django_username()*. If this is not symmetrical to *ldap_to_django_username()*, the behavior is undefined.

CHAPTER 12

Change Log

2.1.1 - 2020-03-26

- Removed drepecated `providing_args` from `Signal` instances.

12.1 2.1.0 - 2019-12-03

- Reject authentication requests without a username.
- Added support for Django 3.0 and Python 3.8.
- Removed support for Django end of life Django 2.1.

12.2 2.0.0 - 2019-06-05

- Removed support for Python 2 and 3.4.
- Removed support for end of life Django 2.0.
- Added support for Django 2.2.
- Add testing and support for Python 3.7 with Django 1.11 and 2.1.
- When *`AUTH_LDAP_SERVER_URI`* is set to a callable, it is now passed a positional `request` argument. Support for no arguments will continue for backwards compatibility but will be removed in a future version.
- Added new *`AUTH_LDAP_NO_NEW_USERS`* to prevent the creation of new users during authentication. Any users not already in the Django user database will not be able to login.

12.3 1.6.1 - 2018-06-02

- Renamed `requirements.txt` to `dev-requirements.txt` to fix Read the Docs build.

12.4 1.6.0 - 2018-06-02

- Updated `LDAPBackend.authenticate()` signature to match Django's documentation.
- Fixed group membership queries with DNs containing non-ascii characters on Python 2.7.
- The setting `AUTH_LDAP_CACHE_TIMEOUT` now replaces deprecated *AUTH_LDAP_CACHE_GROUPS* and *AUTH_LDAP_GROUP_CACHE_TIMEOUT*. In addition to caching groups, it also controls caching of distinguished names (which were previously cached by default). A compatibility shim is provided so the deprecated settings will continue to work.

12.5 1.5.0 - 2018-04-18

- django-auth-ldap is now hosted at https://github.com/django-auth-ldap/django-auth-ldap.
- Removed NISGroupType class. It searched by attribute nisNetgroupTriple, which has no defined EQAULITY rule.
- The python-ldap library is now initialized with `bytes_mode=False`, requiring all LDAP values to be handled as Unicode text (`str` in Python 3 and `unicode` in Python 2), not bytes. For additional information, see the python-ldap documentation on bytes mode.
- Removed deprecated function `LDAPBackend.get_or_create_user()`. Use `get_or_build_user()` instead.

12.6 1.4.0 - 2018-03-22

- Honor the attrlist argument to `AUTH_LDAP_GROUP_SEARCH`
- **Backwards incompatible**: Removed support for Django < 1.11.
- Support for Python 2.7 and 3.4+ now handled by the same dependency, python-ldap >= 3.0.

12.7 1.3.0 - 2017-11-20

- **Backwards incompatible**: Removed support for obsolete versions of Django (<=1.7, plus 1.9).
- Delay saving new users as long as possible. This will allow `AUTH_LDAP_USER_ATTR_MAP` to populate required fields before creating a new Django user.

  `LDAPBackend.get_or_create_user()` is now `get_or_build_user()` to avoid confusion. The old name may still be overridden for now.
- Support querying by a field other than the username field with `AUTH_LDAP_USER_QUERY_FIELD`.
- New method `authenticate_ldap_user()` to provide pre- and post-authentication hooks.
- Add support for Django 2.0.

12.8 1.2.16 - 2017-09-30

- Better cache key sanitizing.

- Improved handling of LDAPError. A case existed where the error would not get caught while loading group permissions.

12.9 1.2.15 - 2017-08-17

- Improved documentation for finding the official repository and contributing.

12.10 1.2.14 - 2017-07-24

- Under search/bind mode, the user's DN will now be cached for performance.

12.11 1.2.13 - 2017-06-19

- Support selective group mirroring with *AUTH_LDAP_MIRROR_GROUPS* and *AUTH_LDAP_MIRROR_GROUPS_EXCEPT*.
- Work around Django 1.11 bug with multiple authentication backends.

12.12 1.2.12 - 2017-05-20

- Support for complex group queries via *LDAPGroupQuery*.

12.13 1.2.11 - 2017-04-22

- Some more descriptive object representations.
- Improved tox.ini organization.

12.14 1.2.9 - 2017-02-14

- Ignore python-ldap documentation and accept `ldap.RES_SEARCH_ENTRY` from `ldap.LDAPObject.result()`.

12.15 1.2.8 - 2016-04-18

- Add *AUTH_LDAP_USER_ATTRLIST* to override the set of attributes requested from the LDAP server.

12.16 1.2.7 - 2015-09-29

- Support Python 3 with pyldap.

12.17 1.2.6 - 2015-03-29

- Performance improvements to group mirroring (from Denver Janke).
- Add `django_auth_ldap.backend.ldap_error` signal for custom handling of `LDAPError` exceptions.
- Add `django_auth_ldap.backend.LDAPBackend.default_settings` for per-subclass default settings.

12.18 1.2.5 - 2015-01-30

- Fix interaction between `AUTH_LDAP_AUTHORIZE_ALL_USERS` and `AUTH_LDAP_USER_SEARCH`.

12.19 1.2.4 - 2014-12-28

- Add support for nisNetgroup groups (thanks to Christopher Bartz).

12.20 1.2.3 - 2014-11-18

- Improved escaping for filter strings.
- Accept (and ignore) arbitrary keyword arguments to `LDAPBackend.authenticate`.

12.21 1.2.2 - 2014-09-22

- Include test harness in source distribution. Some package maintainers find this helpful.

12.22 1.2.1 - 2014-08-24

- More verbose log messages for authentication failures.

12.23 1.2.0 - 2014-04-10

- django-auth-ldap now provides experimental Python 3 support. Python 2.5 was dropped.

  To sum up, django-auth-ldap works with Python 2.6, 2.7, 3.3 and 3.4.

  Since python-ldap isn't making progress toward Python 3, if you're using Python 3, you need to install a fork:

  ```
  $ pip install git+https://github.com/rbarrois/python-ldap.git@py3
  ```

  Thanks to Aymeric Augustin for making this happen.

12.24 1.1.8 - 2014-02-01

- Update *LDAPSearchUnion* to work for group searches in addition to user searches.
- Tox no longer supports Python 2.5, so our tests now run on 2.6 and 2.7 only.

12.25 1.1.7 - 2013-11-19

- Bug fix: *AUTH_LDAP_GLOBAL_OPTIONS* could be ignored in some cases (such as *populate_user()*).

12.26 1.1.5 - 2013-10-25

- Support POSIX group permissions with no gidNumber attribute.
- Support multiple group DNs for *_FLAGS_BY_GROUP.

12.27 1.1.4 - 2013-03-09

- Add support for Django 1.5's custom user models.

12.28 1.1.3 - 2013-01-05

- Reject empty passwords by default.

  Unless *AUTH_LDAP_PERMIT_EMPTY_PASSWORD* is set to True, LDAPBackend.authenticate() will immediately return None if the password is empty. This is technically backwards-incompatible, but it's a more secure default for those LDAP servers that are configured such that binds without passwords always succeed.
- Add support for pickling LDAP-authenticated users.

CHAPTER 13

Contributing

If you'd like to contribute, the best approach is to send a well-formed pull request, complete with tests and documentation. Pull requests should be focused: trying to do more than one thing in a single request will make it more difficult to process.

If you have a bug or feature request you can try logging an issue.

There's no harm in creating an issue and then submitting a pull request to resolve it. This can be a good way to start a conversation and can serve as an anchor point.

13.1 Development

To get set up for development, activate your virtualenv and use pip to install from `dev-requirements.txt`:

```
$ pip install -r dev-requirements.txt
```

To run the tests:

```
$ django-admin test --settings tests.settings
```

To run the full test suite in a range of environments, run tox from the root of the project:

```
$ tox
```

This includes some static analysis to detect potential runtime errors and style issues.

CHAPTER 14

License

Python Module Index

Index