django-auth-ldap Documentation

Release 1.1.8

Peter Sagerson

April 18, 2016

Contents

1 Installation **3**

2 Authentication **5**
2.1 Server Config . . . 5
2.2 Search/Bind . . . 5
2.3 Direct Bind . . . 6
2.4 Notes . . . 6

3 Working With Groups **9**
3.1 Types of Groups . . . 9
3.2 Finding Groups . . . 9
3.3 Limiting Access . . . 10

4 User objects **11**
4.1 User Attributes . . . 11
4.2 Updating Users . . . 12
4.3 Direct Attribute Access . . . 12
4.4 Custom Field Population . . . 12

5 Permissions **15**
5.1 Using Groups Directly . . . 15
5.2 Group Mirroring . . . 15
5.3 Non-LDAP Users . . . 16

6 Multiple LDAP Configs **17**

7 Logging **19**

8 Performance **21**

9 Example Configuration **23**

10 Reference **25**
10.1 Settings . . . 25
10.2 Module Properties . . . 28
10.3 Configuration . . . 29
10.4 Backend . . . 30

11 Change Log **33**

11.1 v1.1.8 - 2014-02-01 . . . 33
11.2 v1.1.7 - 2013-11-19 . . . 33
11.3 v1.1.5 - 2013-10-25 . . . 33
11.4 v1.1.4 - 2013-03-09 . . . 33
11.5 v1.1.3 - 2013-01-05 . . . 33

12 Older Versions **35**

13 License **37**

Python Module Index **39**

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.

This version is officially supported on Python >= 2.5, Django >= 1.3, and python-ldap >= 2.0. It is known to work on earlier versions (especially of Django) and backwards-compatibility is not broken needlessly, however users of older dependencies are urged to test their deployments carefully and be wary of updates.

CHAPTER 1

Installation

This authentication backend enables a Django project to authenticate against any LDAP server. To use it, add *django_auth_ldap.backend.LDAPBackend* to AUTHENTICATION_BACKENDS. Adding *django_auth_ldap* to INSTALLED_APPS is not recommended unless you would like to run the unit tests. LDAP configuration can be as simple as a single distinguished name template, but there are many rich options for working with User objects, groups, and permissions. This backend depends on the python-ldap module.

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. 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
}
```

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 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
```

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. Three built-in subclasses cover most grouping mechanisms:

- *PosixGroupType*
- *MemberDNGroupType*
- *NestedMemberDNGroupType*

posixGroup objects are somewhat specialized, so they get their own class. 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"
```

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_create_user()* if you'd like to override this behavior. See *get_user_model()* if you'd like to substitute a proxy model.

Note: Prior to Django 1.5, user objects were always instances of `User`. Current versions of Django support custom user models via the `AUTH_USER_MODEL` setting. As of version 1.1.4, django-auth-ldap will respect custom user models.

The only required field for a user is the username, which we obviously have. The `User` model is 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'll 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 User Attributes

LDAP directories tend to contain much more information about users that you may wish to propagate. A pair of settings, *AUTH_LDAP_USER_ATTR_MAP* and *AUTH_LDAP_PROFILE_ATTR_MAP*, serve to copy directory information into `User` and profile objects. These are dictionaries that map user and profile model keys, respectively, to (case-insensitive) LDAP attribute names:

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

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": ["cn=staff,ou=groups,dc=example,dc=com",
                 "cn=admin,ou=groups,dc=example,dc=com"],
    "is_superuser": "cn=superuser,ou=groups,dc=example,dc=com"
}

AUTH_LDAP_PROFILE_FLAGS_BY_GROUP = {
    "is_awesome": ["cn=awesome,ou=groups,dc=example,dc=com"]
}
```

If a list of groups is given, the flag will be set if the user is a member of any group.

4.2 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')
```

4.3 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.

4.4 Custom Field Population

If you would like to perform any additional population of user or profile objects, *django_auth_ldap.backend* exposes two custom signals to help: *populate_user* and *populate_user_profile*. These are sent after the

backend has finished populating the respective objects and before they are saved to the database. You can use this to propagate additional information from the LDAP directory to the user and profile objects any way you like.

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_GROUPS* to True. You can also set *AUTH_LDAP_GROUP_CACHE_TIMEOUT* to override the timeout of cache entries (in seconds).

```
AUTH_LDAP_CACHE_GROUPS = True
AUTH_LDAP_GROUP_CACHE_TIMEOUT = 300
```

5.2 Group Mirroring

The second way to turn LDAP group memberships into permissions is to mirror the groups themselves. 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 his LDAP group membership. Note that 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.

The main difference between this approach and *AUTH_LDAP_FIND_GROUP_PERMS* is that *AUTH_LDAP_FIND_GROUP_PERMS* will query for LDAP group membership either for every request or according to the cache timeout. With group mirroring, membership will be updated when the user authenticates. This may not be appropriate for sites with long session timeouts.

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

Logging

LDAPBackend uses the standard 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. Note that this logger is initialized with a level of NOTSET, so you may need to change the level of the logger in order to get debug messages.

```
import logging

logger = logging.getLogger('django_auth_ldap')
logger.addHandler(logging.StreamHandler())
logger.setLevel(logging.DEBUG)
```

CHAPTER 8

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_GROUPS* to `True` to remove most of this traffic. Alternatively, you might consider using *AUTH_LDAP_MIRROR_GROUPS* and relying on `ModelBackend` to supply group permissions.
2. **Don't access user.ldap_user.***. These properties are only cached on a per-request basis. If you can propagate LDAP attributes to a `User` or profile object, they will only be updated at login. `user.ldap_user.attrs` triggers an LDAP connection for every request in which it's accessed. If you're not using *AUTH_LDAP_USER_DN_TEMPLATE*, then accessing `user.ldap_user.dn` will also trigger an LDAP connection.
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 9

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 perhaps:
# 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 = {
    "first_name": "givenName",
    "last_name": "sn",
    "email": "mail"
}

AUTH_LDAP_PROFILE_ATTR_MAP = {
    "employee_number": "employeeNumber"
}

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"
}

AUTH_LDAP_PROFILE_FLAGS_BY_GROUP = {
    "is_awesome": "cn=awesome,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 group memberships for an hour to minimize LDAP traffic
AUTH_LDAP_CACHE_GROUPS = True
AUTH_LDAP_GROUP_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 10

Reference

10.1 Settings

10.1.1 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.

10.1.2 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.

10.1.3 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.

10.1.4 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.

10.1.5 AUTH_LDAP_BIND_PASSWORD

Default: `''` (Empty string)

The password to use with *AUTH_LDAP_BIND_DN*.

10.1.6 AUTH_LDAP_CACHE_GROUPS

Default: `False`

If `True`, LDAP group membership will be cached using Django's cache framework. The cache timeout can be customized with *AUTH_LDAP_GROUP_CACHE_TIMEOUT*.

10.1.7 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.

10.1.8 AUTH_LDAP_DENY_GROUP

Default: `None`

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

10.1.9 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.

10.1.10 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.

10.1.11 AUTH_LDAP_GROUP_CACHE_TIMEOUT

Default: `None`

If *AUTH_LDAP_CACHE_GROUPS* is `True`, this is the cache timeout for group memberships. If `None`, the global cache timeout will be used.

10.1.12 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.

10.1.13 AUTH_LDAP_GROUP_TYPE

Default: `None`

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

10.1.14 AUTH_LDAP_MIRROR_GROUPS

Default: `False`

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 his LDAP groups as Django groups and update his Django group membership to exactly match his LDAP group membership. If the LDAP server has nested groups, the Django database will end up with a flattened representation.

10.1.15 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`.

10.1.16 AUTH_LDAP_PROFILE_ATTR_MAP

Default: `{}`

A mapping from user profile field names to LDAP attribute names. A user's profile will be populated from his LDAP attributes at login.

10.1.17 AUTH_LDAP_PROFILE_FLAGS_BY_GROUP

Default: `{}`

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

10.1.18 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.

10.1.19 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.

10.1.20 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.

10.1.21 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.

10.1.22 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`.

10.1.23 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.

10.1.24 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.

10.2 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.

10.3 Configuration

class django_auth_ldap.config.**LDAPSearch**

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

- base_dn: The distinguished name of the search base.
- scope: One of ldap.SCOPE_*.
- filterstr: 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*)

- searches: 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'*)

- member_attr: 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'*)

10.4 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.**populate_user_profile**
Like *populate_user*, but sent for the user profile object. This will only be sent if the user has an existing profile. As with *populate_user*, it is sent after the backend has finished setting properties and before the object is saved. This signal has two keyword arguments: `profile` is the user profile object and `ldap_user` is the same as `user.ldap_user`. The sender is the *LDAPBackend* class.

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.

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_create_user()* will instantiate. In Django 1.5, custom user models will be respected; in earlier versions, the model defaults to `django.contrib.auth.models.User`. Subclasses would most likely override this in order to substitute a proxy model.

get_or_create_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 the same as `get_or_create()`: an (instance, created) two-tuple.

The default implementation calls `<model>.objects.get_or_create()`, using a case-insensitive query and creating new users with lowercase usernames. The user model is obtained from *get_user_model()*. 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 11

Change Log

11.1 v1.1.8 - 2014-02-01

- Fix #43: 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.

11.2 v1.1.7 - 2013-11-19

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

11.3 v1.1.5 - 2013-10-25

- Fix #41: Support POSIX group permissions with no gidNumber attribute.
- Support multiple group DNs for *_FLAGS_BY_GROUP.

11.4 v1.1.4 - 2013-03-09

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

11.5 v1.1.3 - 2013-01-05

- Fix #33: 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.
- Fix #39: Add support for pickling LDAP-authenticated users.

CHAPTER 12

Older Versions

- django-auth-ldap 1.0.19

CHAPTER 13

License

d

django_auth_ldap, 28
django_auth_ldap.backend, 30
django_auth_ldap.config, 29

Symbols

__init__() (django_auth_ldap.config.ActiveDirectoryGroupType method), 30
__init__() (django_auth_ldap.config.GroupOfNamesType method), 29
__init__() (django_auth_ldap.config.GroupOfUniqueNamesType method), 30
__init__() (django_auth_ldap.config.LDAPGroupType method), 29
__init__() (django_auth_ldap.config.LDAPSearch method), 29
__init__() (django_auth_ldap.config.LDAPSearchUnion method), 29
__init__() (django_auth_ldap.config.MemberDNGroupType method), 29
__init__() (django_auth_ldap.config.NestedActiveDirectoryGroupType method), 30
__init__() (django_auth_ldap.config.NestedGroupOfNamesType method), 29
__init__() (django_auth_ldap.config.NestedGroupOfUniqueNamesType method), 30
__init__() (django_auth_ldap.config.NestedMemberDNGroupType method), 29
__init__() (django_auth_ldap.config.NestedOrganizationalRoleGroupType method), 30
__init__() (django_auth_ldap.config.OrganizationalRoleGroupType method), 30
__init__() (django_auth_ldap.config.PosixGroupType method), 29

A

ActiveDirectoryGroupType (class in django_auth_ldap.config), 30
AUTH_LDAP_ALWAYS_UPDATE_USER
    setting, 25
AUTH_LDAP_AUTHORIZE_ALL_USERS
    setting, 25
AUTH_LDAP_BIND_AS_AUTHENTICATING_USER
    setting, 25
AUTH_LDAP_BIND_DN
    setting, 25
AUTH_LDAP_BIND_PASSWORD
    setting, 25
AUTH_LDAP_CACHE_GROUPS
    setting, 26
AUTH_LDAP_CONNECTION_OPTIONS
    setting, 26
AUTH_LDAP_DENY_GROUP
    setting, 26
AUTH_LDAP_FIND_GROUP_PERMS
    setting, 26
AUTH_LDAP_GLOBAL_OPTIONS
    setting, 26
AUTH_LDAP_GROUP_CACHE_TIMEOUT
    setting, 26
AUTH_LDAP_GROUP_SEARCH
    setting, 26
AUTH_LDAP_GROUP_TYPE
    setting, 27
AUTH_LDAP_MIRROR_GROUPS
    setting, 27
AUTH_LDAP_PERMIT_EMPTY_PASSWORD
    setting, 27
AUTH_LDAP_PROFILE_ATTR_MAP
    setting, 27
AUTH_LDAP_PROFILE_FLAGS_BY_GROUP
    setting, 27
AUTH_LDAP_REQUIRE_GROUP
    setting, 27
AUTH_LDAP_SERVER_URI
    setting, 27
AUTH_LDAP_START_TLS
    setting, 28
AUTH_LDAP_USER_ATTR_MAP
    setting, 28
AUTH_LDAP_USER_DN_TEMPLATE
    setting, 28
AUTH_LDAP_USER_FLAGS_BY_GROUP
    setting, 28
AUTH_LDAP_USER_SEARCH
    setting, 28

D

django_auth_ldap (module), 28
django_auth_ldap.backend (module), 30
django_auth_ldap.config (module), 29
django_to_ldap_username() (django_auth_ldap.backend.LDAPBackend method), 31

G

get_or_create_user() (django_auth_ldap.backend.LDAPBackend method), 31
get_user_model() (django_auth_ldap.backend.LDAPBackend method), 31
GroupOfNamesType (class in django_auth_ldap.config), 29
GroupOfUniqueNamesType (class in django_auth_ldap.config), 30

L

ldap_to_django_username() (django_auth_ldap.backend.LDAPBackend method), 31
LDAPBackend (class in django_auth_ldap.backend), 30
LDAPBackend.settings_prefix (in module django_auth_ldap.backend), 30
LDAPGroupType (class in django_auth_ldap.config), 29
LDAPSearch (class in django_auth_ldap.config), 29
LDAPSearchUnion (class in django_auth_ldap.config), 29

M

MemberDNGroupType (class in django_auth_ldap.config), 29

N

NestedActiveDirectoryGroupType (class in django_auth_ldap.config), 30
NestedGroupOfNamesType (class in django_auth_ldap.config), 29
NestedGroupOfUniqueNamesType (class in django_auth_ldap.config), 30
NestedMemberDNGroupType (class in django_auth_ldap.config), 29
NestedOrganizationalRoleGroupType (class in django_auth_ldap.config), 30

O

OrganizationalRoleGroupType (class in django_auth_ldap.config), 30

P

populate_user (in module django_auth_ldap.backend), 30
populate_user() (django_auth_ldap.backend.LDAPBackend method), 30
populate_user_profile (in module django_auth_ldap.backend), 30
PosixGroupType (class in django_auth_ldap.config), 29

S

setting
AUTH_LDAP_ALWAYS_UPDATE_USER, 25
AUTH_LDAP_AUTHORIZE_ALL_USERS, 25
AUTH_LDAP_BIND_AS_AUTHENTICATING_USER, 25
AUTH_LDAP_BIND_DN, 25
AUTH_LDAP_BIND_PASSWORD, 25
AUTH_LDAP_CACHE_GROUPS, 26
AUTH_LDAP_CONNECTION_OPTIONS, 26
AUTH_LDAP_DENY_GROUP, 26
AUTH_LDAP_FIND_GROUP_PERMS, 26
AUTH_LDAP_GLOBAL_OPTIONS, 26
AUTH_LDAP_GROUP_CACHE_TIMEOUT, 26
AUTH_LDAP_GROUP_SEARCH, 26
AUTH_LDAP_GROUP_TYPE, 27
AUTH_LDAP_MIRROR_GROUPS, 27
AUTH_LDAP_PERMIT_EMPTY_PASSWORD, 27
AUTH_LDAP_PROFILE_ATTR_MAP, 27
AUTH_LDAP_PROFILE_FLAGS_BY_GROUP, 27
AUTH_LDAP_REQUIRE_GROUP, 27
AUTH_LDAP_SERVER_URI, 27
AUTH_LDAP_START_TLS, 28
AUTH_LDAP_USER_ATTR_MAP, 28
AUTH_LDAP_USER_DN_TEMPLATE, 28
AUTH_LDAP_USER_FLAGS_BY_GROUP, 28
AUTH_LDAP_USER_SEARCH, 28

V

version (in module django_auth_ldap), 28
version_string (in module django_auth_ldap), 28