django-windowsauth

Dan Yishai

Dec 19, 2020

INSTALLATION AND SETUP

Easy integration and deployment of Django projects into Windows Systems.

CHAPTER

ONE

FEATURES

- Deploy to Microsoft IIS quickly using wfastcgi and `createwebconfig` command
- Authenticate via IIS's Windows Authentication
- Authorize against Active Directory using ldap3 package
- Manage LDAP connections for easy integrations
- (Coming soon) Debug using django-debug-toolbar

1.1 Quick Start

1. Install with `pip install django-windowsauth`
2. Run `py manage.py migrate windows_auth`
3. Add "fastcgi application" with `wfastcgi-enable`
4. Configure project settings:

```
INSTALLED_APPS = [
   "windows_auth",
]

MIDDLEWARE = [
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.RemoteUserMiddleware',
]

AUTHENTICATION_BACKENDS = [
    "windows_auth.backends.WindowsAuthBackend",
    "django.contrib.auth.backends.ModelBackend",
]

WAUTH_DOMAINS = {
    "<your domain's NetBIOS Name> (EXAMPLE)": {
        "SERVER": "<domain FQDN> (example.local)",
        "SEARCH_BASE": "<search base> (DC=example,DC=local)",
        "USERNAME": "<bind account username>",
        "PASSWORD": "<bind account password>",
    }
}

# optional
```

(continues on next page)

(continued from previous page)

```
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / "static"

MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / "media"
```

5. Generate **web.config** files with `py manage.py createwebconfig -s -m -w`.
6. Create new IIS Website from the project files

1.2 Installation and Setup

This is a **detailed** walk-through the *django-windowsauth* installation and setup process.
For easy and quick installation please refer to the *Quick Start* guide.

1.2.1 Install and Setup IIS

First, you may need to install IIS role. This can be done though the Control Panel > Add and Remove Programs > Install Features (`appwiz.cpl`) or via **Server Manager**.

Those are the features you should select:

1. Application / CGI
2. Security / Windows Authentication
3. (suggested) Performance Features / Dynamic Content Compression
4. (suggested) Health and Diagnostics / Request Monitor
5. (suggested) Health and Diagnostics / Tracing

Next you will need to unlock some configuration section to later use the `createwebconfig` management command.

To unlock configuration sections:

1. Open IIS Manager > Configuration Editor
2. Select section `system.webServer/handlers`
3. Click `Unlock section` on the right sidebar.
4. Repeat for sections `system.webServer/security/authentication/anonymousAuthentication` and `system.webServer/security/authentication/windowsAuthentication`.

Note: For more information visit the IIS Topic on Microsoft Docs: https://docs.microsoft.com/en-us/iis

1.2.2 Getting it

You can get django-windowsauth by using pip:

```
$ pip install django-windowsauth
```

If you want to install it from source, grab the git repository and run setup.py:

```
$ git clone https://github.com/danyi1212/django-windowsauth.git
$ python setup.py install
```

1.2.3 Installing

You will need to add the `windows_auth` application to the INSTALLED_APPS setting in you Django project settings file.

```
INSTALLED_APPS = [
    ...
    'windows_auth',
    ...
]
```

This will allow to execute the `createwebconfig` command, add the new model *LDAPUer* and register it's Django Admin page.

Next, you will need to run the `migrate` management command to create the new SQL table of the new models.:

```
$ py manage.py migrate windows_auth
```

Note: This will perform migrations only for **windows_auth** app. If you have other migrations pending, you may want to omit the **windows_auth** argument to perform all available migrations.

1.2.4 Configure

In order to receive correctly the authenticated user from the **IIS Windows Authentication**, you will need to add a middleware called `RemoteUserMiddleware`. This middleware must be after `AuthenticationMiddleware`, that is usually provided by default with Django's `startproject` template.

```
MIDDLEWARE = [
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.RemoteUserMiddleware',
    ...
]
```

To process the information passed from the **IIS Windows Authentication** and translate it into a **Django User**, you will need to specify the `WindowsAuthBackend` authentication backend.

```
AUTHENTICATION_BACKENDS = [
    'windows_auth.backends.WindowsAuthBackend',
    'django.contrib.auth.backends.ModelBackend',
]
```

Note: Be aware, this configuration keeps the Django's default **ModelBackend** in order to allow for fallback to **Django Native Users**. It can be used to authenticate without IIS, when using the `runserver` management command for example.

This is usually not advised to configure for **Production** setups, but only for **Development**.

See also:

Django documentation about *Authenticating using REMOTE_USER* https://docs.djangoproject.com/en/3.1/howto/auth-remote-user/

Next you will need to configure the settings for your **Domain** to allow for LDAP integration with **Active Directory**.

```
WAUTH_DOMAINS = {
    "EXAMPLE": {  # this is your domain's NetBIOS Name, same as in "EXAMPLE\\username
↪" login scheme
        "SERVER": "example.local",  # the FQDN of the DC server, usually is the FQDN␣
↪of the domain itself
        "SEARCH_BASE": "DC=example,DC=local",  # the default Search Base to use when␣
↪searching
        "USERNAME": "EXAMPLE\\bind_account",  # username of the account used to␣
↪authenticate your Django project to Active Directory
        "PASSWORD": "<super secret>",  # password for the binding account
    }
}
```

See also:

About LDAP Search Base: https://docs.microsoft.com/en-us/windows/win32/ad/binding-to-a-search-start-point

(optionally) Configure **file path** and **url path** settings for your `static` and `media` files.

```
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / "static"

MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / "media"
```

You may need to execute `$ py manage.py collectstatic` management command after modifying the `STATIC_ROOT` setting.

See also:

Full how-to guide to *Serve Static Files through IIS*

1.2.5 Setup Logging

Throughout this whole module, logging is done to logger named `wauth`. You may handle and configure this logger through Django's setting `LOGGING`.

This can be done by adding the logger like so:

```
'wauth': {
    'handlers': ['console', 'file', 'mail_admins'],
    'level': 'INFO',
    'propagate': False,
},
```

Additionally, you may want to configure logging for `ldap3`. You can add this logger:

```
'ldap3': {
    'handlers': ['console', 'ldap'],
    'level': 'DEBUG',
    'propagate': False,
}
```

And make sure to configure `ldap3` log type, like this:

```
from ldap3.utils.log import set_library_log_detail_level, BASIC
set_library_log_detail_level(BASIC)
```

The lines above can be added in your Django settings file, just after the `LOGGING` setting. Remember to document about that in your code!

See also:

More information of that on https://ldap3.readthedocs.io/en/latest/logging.html

For your convenience, those are the handles used in the examples above:

```
'handlers': {
    'console': {
        'class': 'logging.StreamHandler',
        'level': 'WARNING',
    },
    'file': {
        'level': 'INFO',
        'class': 'logging.handlers.RotatingFileHandler',
        'maxBytes': 2 ** 20 * 100,  # 100MB
        'backupCount': 10,
        'filename': BASE_DIR / 'logs' / 'debug.log',
    },
    'ldap': {
        'level': 'INFO',
        'class': 'logging.handlers.RotatingFileHandler',
        'maxBytes': 2 ** 20 * 100,  # 100MB
        'backupCount': 10,
        'filename': BASE_DIR / 'logs' / 'ldap.log',
    },
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'include_html': True,
    },
},
```

Note: You will need to configure settings for sending emails to use the `mail_admins` handler: https://docs.djangoproject.com/en/3.1/topics/email/

1.2.6 Publish to IIS

First, we will need to create the `web.config` files for the IIS Website configuration. This can be done simply by running the management command::

```
$ py manage.py createwebconfig -s -m -w
```

Notice the `-s` and `-m` switches, those are to add configurations for **Serving Static Files though IIS**. You may want to omit those switches if you are not planning to serve static files though IIS.

The `-w` parameter configures IIS's `Windows Authentication` and disables `Anonymous Authentication` in the `web.config` file. You may want to change those settings manually to avoid **unlocking those configuration sections**.

See also:

Reference for `createwebconfig` at *Management Commands*

Next you will need to create a new IIS Website for your Django Project.

1. Open **IIS Manager**
2. Right-click over **sites**
3. Click **Add website...**
4. Give a **name** for your site (should use the same as for your Django project)
5. Specify **Physical path** for the root of your Django project folder (where the `manage.py` is)
6. Provide **binding information** as needed (can be changed later)

Congratulation, now you should be able to browse to your new website!

Next are some things to setup and verify before publishing to production...

1.3 Publish to Production

This tutorial is still in the process of writing...

1.4 Serve Static Files through IIS

Generally websites have static files such as CSS, JS, Images served to clients beside the primary responses. Those files are considered as "Static Files" because they can be delivered without being generated, modified or processed.

In Django, static files can be served by the Django Framework itself. This is very convenient during **Development**, but is not suitable for **Production** use.

See also:

About Serving Static Files: https://docs.djangoproject.com/en/3.1/howto/static-files/

For production use, it is advised to let the **Web Server** to serve the Static Files. This is how it can be done:

Note: This how-to describes serving both **Static Files** and **Media Files**. In case you don't need or use one of those features, you can just ignore the respective parts in the tutorial.

First you will need to configure the following settings:

```
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / "static"

MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / "media"
```

The `STATIC_URL` represents the file path **over HTTP**, while the `STATIC_ROOT` directs to the **Physical path** of the files in the Web Server's OS. Meanwhile from the **IIS** point of view, the **HTTPS path** is derived from the file's **Physical path** location. Although this can be altered using **Virtual Directories**, it is usually advised not to.

The same applies for the `MEDIA_URL` and `MEDIA_ROOT` settings.

Next we will need to create `web.config` files in each folder to configure IIS to server Static Files.

Note: Any time the `STATIC_ROOT` setting is changes, you will need to start over from this step.

This can be done by running the `createwebconfig` management command::

```
$ py manage.py createwebconfig -s -m
```

The `-s` switch is used configure the `STATIC_ROOT` folder, while `-m` switch is used to configure the `MEDIA_ROOT` folder.

Now all we need to do is to **collect** all the static files from the many Django apps into the `STATIC_ROOT` folder. This can be done by running the `collectstatic` management command::

```
$ py manage.py collectstatic
```

See also:

About `collectstatic` command: https://docs.djangoproject.com/en/3.1/ref/contrib/staticfiles/#django-admin-collectstatic

At this point, in case you have configured the **URL path** and **Physical path** the same, the Web Server should serve all static files correctly.

In case you have configured **different paths**, you will probably want to setup **Virtual Directories**.

This can be useful when you want to store Static and / or Media file **outside** the Django project's folder (the website's root folder), on a separate disk for example.

To create the Virtual Directories:

1. Open **IIS Manager**
2. Right-click on **your website**
3. Click **"Add Virtual Directory..."**
4. Set the **"Alias"** for the same value as `STATIC_URL` setting
5. Set the **"Physical Path"** for the same value `STATIC_ROOT` setting

You may do the same with the `MEDIA_URL` and `MEDIA_ROOT` settings in order to add Virtual Directory for serving **Media Files**.

See also:

Microsoft Docs on IIS Virtual Directories https://docs.microsoft.com/en-us/iis/get-started/planning-your-iis-architecture/understanding-sites-applications-and-virtual-directories-on-iis#virtual-directories

1.5 Using Custom User Model Mappings

This tutorial is still in the process of writing...

1.6 Utilizing the LDAP Manager throughout Django

This tutorial is still in the process of writing...

1.7 Managing Secrets

This tutorial is still in the process of writing...

1.8 Securing LDAP Connections

This tutorial is still in the process of writing...

1.9 Customize Error Pages

This tutorial is still in the process of writing...

1.10 Debug with django-debug-toolbar

This feature is still not available in this version...

1.11 Settings

1.11.1 WAUTH_USE_SPN

Type `bool`; Default to `False`; Not Required.
Expect the REMOTE_USER header value to be in Windows SPN username scheme.

By default, IIS will present the authenticated user by it's Down-Level Logon Name, for example "EXAMPLE\username". Setting this value to `True` will will expect the authenticated user to be presented by it's User Principal Name, for example "username@example.local".

Note: When using `SPN` the domain of the authenticated user will be detected by the **Domain's FQDN** instead of it's **NetBIOS Name**!

This means that you will need to configure `WAUTH_DOMAINS` by created with the FQDN of their domain, and not their NetBIOS Name. This is also means all new LDAPUser domain values will be FQDNs and not NetBIOS Names

If you are planning to migrate between using Down-Level to SPN, first of all **don't**. In case you still need to switch between them, you can either **manually replace** the LDAPUser's domain values from the old NetBIOS Names to the new FQDNs, or just **delete** all LDAPUsers and let them be created again when a user login again after change.

1.11.2 WAUTH_DOMAINS (Required)

Type `dict`; Default to `None`; Required.
LDAP Settings for each domain.

Dictionary of domain NetBIOS Names and their settings for LDAP connection. Domain LDAP Settings can be written as a dictionary with the settings in UPPERCASE and their values, or as an `LDAPSettings` object.

A default domain settings can be used as a fallback settings for requested domains that are missing from `WAUTH_DOMAINS` by using **"__default__"** as the domain name. When using only the default domain settings, you may want to specify manually the `WAUTH_PRELOAD_DOMAINS` setting.

Each of the domain settings can be configured as a **function** that will be used as callback when accessing the setting and be called with the **domain as it first argument**. This can be used with `lambda` functions for lazy setting values.

See also:

More information about domain LDAP Settings can be found at *LDAP Settings* reference.

1.11.3 WAUTH_RESYNC_DELTA

Type `timedelta`, `str`, `int` or `None`; Default to `timedelta(minutes=10)`; Not Required.
Minimum time (seconds) until automatic re-sync user's fields and permissions against LDAP.

Configure when to **automatically synchronize** the user's fields and groups (and permissions) against Active Directory via LDAP. On each request the user makes, if the user **haven't synchronized** in the time specified, the `WindowsAuthBackend` attempt to perform synchronization again on the user. This is used to make sure the user permissions and properties match those in Active Directory.

The value is used as **number of seconds** in `int`, `str` or any other object that can be casted to `int`. The value can also be a `django.utils.timezone.timedelta` object.

In case you need to synchronize the user on every request, you can configure the setting to `0`.
To disable automatic synchronizations via LDAP, you can configure the setting to `None` or `False`.

Note: Synchronizing user via LDAP can delay the Request / Response processing by only few ms, but your experience may vary. You can debug your setup using *Debug with django-debug-toolbar*.

1.11.4 WAUTH_USE_CACHE

Type `boot`; Default to `DEBUG`, otherwise `False`; Not Required.
Use cache backend instead of DB for determining user re-sync.

When using user automatic synchronization, the check whether user requires a re-sync is verified against the `LDAPUser` model and it requires an SQL Query.

To avoid this query and allow for better performance, this setting can allow you to use Django's cache framework instead of the default model verification against the DB. This will require you to setup your cache backend in setting `CACHES`.

In production, it is advised to use the cache setting instead of the default model based verification.

1.11.5 WAUTH_REQUIRE_RESYNC

Type `boot`; Default to `False`; Not Required.
Raise exception and return Error 500 when user failed to synced to domain.

When using user automatic synchronization, propagate any exception raised during synchronization. This will result with the user receiving a **Error 500** when they fail to synchronize properly.

This is useful for security sake, when **requiring** users to have the most updated fields and permissions.

While developing in debug, it is usually useful to **receive information** about the synchronization exception.

Note: In any case, the synchronization exception **will be logged** as error with the exception information included. If you have setup logging and email reporting for server admins, you can also **receive the exception details by email**.

See the documentation about ../installation/logging

1.11.6 WAUTH_LOWERCASE_USERNAME

Type `boot`; Default to `True`; Not Required.
Lowercase the username to mimic non-case sensitive LDAP backends like Active Directory.

Windows systems, like Active Directory are **non-case sensitive**. While python, Django, and most Databases are **case sensitive**, you can lower case every username to **mimic** the non-case sensitive behavior of the Windows system.

1.11.7 WAUTH_IGNORE_SETTING_WARNINGS

Type `boot`; Default to `True`; Not Required.
Skip verification of domain settings on server startup.

By default, on every startup of you Django project the settings are validated.

This setting can be used to ignore the warnings raised by detecting users with domains missing from settings in `WAUTH_DOMAINS`, and **Unknown Settings** detected in domain LDAP Settings.

1.11.8 WAUTH_PRELOAD_DOMAINS

Type `tuple` or `bool`; Default to `None`; Not Required.
List of domains to preload and connect during Django project startup

LDAP Connections are **cached in process memory** to retain connections for multiple request / response cycles. This setting lists the domains to preload, connection and bind during you **Django project startup**. This way, the first request for a process will not have wait extra time for the LDAP connection to load and connect.

When the setting is configured to `None` or `True`, all the domains configured in `WAUTH_DOMAINS` settings are **preloaded**. In case you use only the **default domain settings** in the `WAUTH_DOMAINS` setting, it is advised to **manually** configure this setting to preload the relevant domains.

To enable LDAP Connection **lazy loading**, you can set this setting to `False`.

Note: When using `runserver` command, due to the server first **validating models** before loading the project, it may seam like **multiple connections** get initiated for the same domains.

By setting this setting, it may cause **multiple LDAP connections** to be established and terminate quickly for each domain.

You should **not be warned** by this behavior as this is behaves like a **quick connection test** to your LDAP server, and this is should only happened during **development phase**. In case you would like to **avoid this behavior** anyway, you can use the `runserver --noreload` parameter, or modifying the `WAUTH_PRELOAD_DOMAINS` setting to `False` when debugging.

1.12 LDAP Settings

LDAP Settings are the settings used to configure **LDAP connection** to domains. They are configured inside the `WAUTH_DOMAINS` setting of your Django project settings file, as the **value** for each domain key.

1.12.1 Configuring

LDAP Settings can be represented as a regular **Python Dictionary**, like this:

```
WAUTH_DOMAINS = {
    "EXAMPLE": {
        "SERVER": "example.local",
        "SEARCH_BASE": "DC=example,DC=local",
        "USERNAME": "EXAMPLE\\bind_account",
        "PASSWORD": "*********",
    }
}
```

Or as an LDAPSettings object, like this:

```
WAUTH_DOMAINS = {
    "EXAMPLE": LDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
        USERNAME="EXAMPLE\\bind_account",
```

(continues on next page)

(continued from previous page)

```
        PASSWORD="*********",
    ),
}
```

When using a **Python Dictionary**, each setting can be configured to a **callback function** that will be called with the specified domain as first and only argument. For example:

```
WAUTH_DOMAINS = {
    "EXAMPLE": {
        "USERNAME": lambda domain: f"{domain}\\bind_account",
    }
}
```

1.12.2 Using defaults

Sometimes when using multiple domains it is easier to configure settings **globally** or to **specify defaults** for unanticipated domains.

When configuring LDAP Settings as a **Python Dictionary**, this can be done by using the `"__default__"` key in `WAUTH_DOMAINS` settings. Every setting configured in the `"__default__"`, and are **not configured explicitly** for the domain, in will propagate. For example:

```
WAUTH_DOMAINS = {
    "__default__": {
        "USE_SSL": True,
    },
    "EXAMPLE1": {
        "SERVER": "example.local",
        "SEARCH_BASE": "DC=example,DC=local",
        "USERNAME": "EXAMPLE\\bind_account",
        "PASSWORD": "*********",
        "USE_SSL": False,
    },
    "EXAMPLE2": {
        "SERVER": "example.local",
        "SEARCH_BASE": "DC=example,DC=local",
        "USERNAME": "EXAMPLE\\bind_account",
        "PASSWORD": "*********",
    }
}
```

In this case, `EXAMPLE1` will have `USE_SSL = False` and `EXAMPLE2` will have `USE_SSL = True`.

When using `LDAPSettings` objects, this can be done by inheriting and creating a custom `LDAPSettings` class. For example:

```
@dataclass()
class MyLDAPSettings(LDAPSettings):
    USE_SSL: bool = False


WAUTH_DOMAINS = {
    "EXAMPLE": MyLDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
```

(continues on next page)

(continued from previous page)

```
        USERNAME="EXAMPLE\\bind_account",
        PASSWORD="*********",
    ),
}
```

1.12.3 Extending LDAP Settings

Sometimes it is useful to have some **extra LDAP Settings** for use with the LDAP Manager.

It is possible to create a custom `LDAPSettings` class and use it to configure the LDAP Settings for domains. Those extra setting will be available in the **settings attribute** of `LDAPManager` objects, and can be used **throughout your code**. Those settings should not affect the existing settings used by `django-windowsauth` for User synchronization or any other uses.

Custom LDAP Settings objects can be created by inheriting from the `LDAPSettings` dataclass, like so:

```
@dataclass()
class MyLDAPSettings(LDAPSettings):
    EXTRA_SETTING: str = "Hello, world!"


WAUTH_DOMAINS = {
    "EXAMPLE": MyLDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
        USERNAME="EXAMPLE\\bind_account",
        PASSWORD="*********",
    ),
}
```

Then the setting could be accessed from `LDAPManager` object:

```
>>> from windows_auth.ldap import get_ldap_manager
>>> manager = get_ldap_manager("EXAMPLE")
>>> manager.settings.EXTRA_SETTING
"Hello, world!"
```

1.12.4 Base Settings

SERVER

Type `bool`; **Required**.
FQDN, IP, or URL of the LDAP Server.

The Fully Qualified Domain Name, IP Address or complete URL in the scheme `scheme://hostname:hostport` of the LDAP Server. This setting will be used as `host` property for ldap3's Server object.

When using Active Directory, this address should direct to a DC Server (Domain Controller) for the domain. By default, the FQDN of the domain itself will be resolved into your current configured DC Server. That way, in case you have multiple DC servers in your domain, you will be dynamically changing the server you are accessing.

See also:

From the Microsoft Docs https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/plan/domain-controller-location

USERNAME

Type `str`; **Required**.
The account to be used when binding to the LDAP Server.

The username in one of the Credentials Manager API's formats (Down-Level or SPN) of a user with the permissions needed for your application. By default, **read-only permissions** for the user accounts that are able to authenticate via IIS Windows Authentication to **your website** is needed.

If you are planing to use **NTLM authentication** to your LDAP Server, the username must be in the Down-Level Logon Name format (DOMAIN\username).

In production, it is advised to use a **dedicated Service Account** to authorize your application in your Active Directory domain.

PASSWORD

Type `str`; **Required**.
Password of the user to be used when binding to the LDAP Server.

The password for the user used to authenticate to the LDAP Server.

> **Warning:** It is highly advised not to store sensitive secrets like password in your code. You should use a safe and secure place to store the password. See the tutorial manage_secrets

SEARCH_BASE

Type `str`; **Required**.
The DN of the container used as starting point for LDAP searches.

When querying LDAP Directories, it is required to specify the **root container** to start the search from. Then, depending on the search scope, the objects are searched **directly or indirectly** in respect to the search base container.

For searches throughout all the domain's containers, the search base DN is usually in the format `DC=<domain name>,DC=<parent domain>`.

See also:

Microsoft docs about search bases https://docs.microsoft.com/en-us/windows/win32/ad/binding-to-a-search-start-point

USE_SSL

Type `bool`; Default to `True`; Not Required.
Connect to LDAP over secure port, usually 636.

This setting is used as the `use_ssl` parameter for the ldap3 `Server` object.

See also:

ldap3 Server object docs https://ldap3.readthedocs.io/en/latest/server.html

READ_ONLY

Type `bool`; Default to `True`; Not Required.
Prevent modify, delete, add and modifyDn (move) operations.

Connect to the LDAP Server with a read only protection. This can farther minimize risks and vulnerabilities from unwanted operations against the LDAP Server.

This setting is used as the `read_only` parameter for the ldap3 `Connection` object.

> **Warning:** This is not guaranteed to be a risk / vulnerabilities free connection, **make sure to minimize the bind account's permissions**

SERVER_OPTIONS

Type `dict`; Default to `{}`; Not Required.
Extra parameters for the ldap3 `Server` object.

A dictionary of extra keyword arguments to pass when creating the ldap3 `Server` object.

See also:

For more information, see ldap3 docs https://ldap3.readthedocs.io/en/latest/server.html

CONNECTION_OPTIONS

Type `dict`; Default to `{}`; Not Required.
Extra parameters for the ldap3 `Connection` object.

A dictionary of extra keyword arguments to pass when creating the ldap3 `Connection` object.

See also:

For more information, see ldap3 docs https://ldap3.readthedocs.io/en/latest/connection.html

PRELOAD_DEFINITIONS

Type `tuple`; Default is shown below; Not Required.
Preload LDAP schema for defining LDAP objects in Python.

A list of LDAP Object definitions to **preload** while connecting to the LDAP Server. This **caches** ldap3 `ObjectDef` objects on the `LDAPManager` object for each defined object class. The object definitions are later get used for **parsing the objects** received from querying the LDAP Directory. Preloading the object definitions can **minimize the extra delay** for first query for an object.

The definitions can be listed as a **simple string** referring to an LDAP object class, or a **2 valued tuple** with the LDAP object class string on the first value, and a list of **extra attributes** on the second value. For example:

```
{
    "PRELOAD_DEFINITIONS": (
        ("user", ["sAMAccountName"]),
        "group"
    ),
}
```

The configuration above is the actual default configuration for this setting.

USER_FIELD_MAP

Type `dict`; Default is shown below; Not Required.
Translate User Model fields to LDAP User object attributes.

Provide a mapping for your **Django User Model fields** to the **LDAP User object attributes**. Those mappings are used when synchronizing Django Users to their related LDAP Users.

In case you using a **Custom User Model** in your Django project, you also will be able to map them to LDAP Attributes. This is mentioned in the tutorial *Using Custom User Model Mappings*.

Note: Make sure to specify the needed attributes when **preloading definitions** for non-default attributes.

```
{
    "USER_FIELD_MAP": {
        "username": "sAMAccountName",
        "first_name": "givenName",
        "last_name": "sn",
        "email": "mail",
    }
}
```

The configuration above is the actual default configuration for this setting.

USER_QUERY_FIELD

Type `str`; Default to `username`; Not Required.
The User Model field used for searching the related LDAP User object.

When synchronizing users to LDAP, they are first need to be searched. This setting can allow you to specify the **Django User Model field** that will be compared to the related **LDAP Attribute** using the `USER_FIELD_MAP` setting when searching for the related user.

This setting may be useful when using a **Custom User Model** in your Django project. This is mentioned in the tutorial *Using Custom User Model Mappings*.

Note: Make sure to use a unique field, that is unique at the **LDAP side** too. If multiple objects are found, the synchronization will fail.

GROUP_ATTRS

Type `str` or `tuple`; Default to `cn`; Not Required.
The LDAP group attributes to search when matching to Django groups.

When synchronizing users against LDAP, you can **replicate group memberships**. When used, you may want to specify what **LDAP attributes** are used when comparing the **Django Group's names** to LDAP Groups.

This setting can be a **single string** for comparing a single attribute, or a **tuple** for comparing multiple attributes. When comparing multiple attributes, if one of them matches the Django Group's name, the user is added to that group.

> **Warning:** The comparing is done on the **Python side** by the ldap3 library. Using many attributes to search groups may result in **longer synchronization times**.

SUPERUSER_GROUPS

Type `tuple` or `str`; Default to `Domain Admins`; Not Required.
LDAP Groups to check membership for setting Django User's "is_superuser" flag.

When synchronizing users against LDAP, you can specify a **list of LDAP Groups** to match for setting the Django User's `is_superuser` flag. If the user is member in **one** of the listed LDAP groups, the `is_superuser` flag will be set to `True`, otherwise it is set to `False`.

Configuring this setting to `None` will not modify the `is_superuser` flag.
Configuring this setting to a **string** is equal to a **single length tuple**.

The group membership is checked by comparing the **groups listed in this setting** to the **LDAP Group Attributes** listed in `GROUP_ATTRS` setting.

STAFF_GROUPS

Type `tuple` or `str`; Default to `Administrators`; Not Required.
LDAP Groups to check membership for setting Django User's "is_staff" flag.

When synchronizing users against LDAP, you can specify a **list of LDAP Groups** to match for setting the Django User's `is_staff` flag. If the user is member in **one** of the listed LDAP groups, the `is_staff` flag will be set to `True`, otherwise it is set to `False`.

Configuring this setting to `None` will not modify the `is_staff` flag.
Configuring this setting to a **string** is equal to a **single length tuple**.

The group membership is checked by comparing the **groups listed in this setting** to the **LDAP Group Attributes** listed in `GROUP_ATTRS` setting.

ACTIVE_GROUPS

Type `tuple` or `str`; Default to `None`; Not Required.
LDAP Groups to check membership for setting Django User's "is_active" flag.

When synchronizing users against LDAP, you can specify a **list of LDAP Groups** to match for setting the Django User's `is_active` flag. If the user is member in **one** of the listed LDAP groups, the `is_active` flag will be set to `True`, otherwise it is set to `False`.

Configuring this setting to `None` will not modify the `is_active` flag.
Configuring this setting to a **string** is equal to a **single length tuple**.

The group membership is checked by comparing the **groups listed in this setting** to the **LDAP Group Attributes** listed in `GROUP_ATTRS` setting.

GROUP_MAP

Type `dict`; Default is `{}`; Not Required.
Map one or more LDAP Groups membership to Django Group membership.

When synchronizing users against LDAP, you can specify a mapping of LDAP Groups to Django Groups. If the user is member in **one** of the listed LDAP groups, it will be added to the respective Django Group.

The setting is configured as a dictionary where the **keys are Django Groups names** and the value is **a string or a list of LDAP Groups**. The LDAP Groups are compared using the attributes listed in the `GROUP_ATTRS` setting.

When a user synchronizes, LDAP group membership will be checked **directly or in-directly** for at least one of the listed groups. For each LDAP group that the user is found to be a member of, it will be **added** to the related Django group, otherwise it will be **removed** from it. Groups that are **not listed** in this setting will **not be affected** by this.

Warning: When a group that is configured in this setting is missing, it will be **created automatically**.

1.13 Authentication Backend

This reference is not available yet...

1.14 Models

1.14.1 LDAPUser

Used to same user's domain information and perform domain related actions.

Fields:

- **user** - One to one relation for user model using `get_user_model` function.
- **domain** - User's domain name (usually) as NetBIOS Name.

Methods:

- **get_ldap_manager()** - Get `LDAPManager` for user's domain.
- **get_ldap_attr(attribute, as_list)** - Get LDAP attribute of the related LDAP user.
- **get_ldap_user()** - Get related LDAP user as ldap3 `Entry` object.
- **get_ldap_groups()** - get LDAP Reader for all groups the user is a member of.
- **sync()** - Synchronize Django user to related LDAP User.

The `LDAPUser` for a Django User can be accessed via `user.ldap`. For example, you can trigger sync with `request.user.ldap.sync()`, or display the user's Windows Logon Name with `request.user.ldap`.

Note: The `LDAPUser` is represented by the **Down-level Logon Name** or **SPN** determined by the `WAUTH_USE_SPN` setting. More on that in the *Settings*.

1.15 Management Commands

1.15.1 createwebconfig

This reference is not available yet...